1985

The animation of algorithms :

Peter J. Floriani
Lehigh University

THE ANIMATION OF ALGORITHMS

Theory and Practice

by

Peter J. Floriani

A Thesis

Presented to the Graduate Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Computer Science

Lehigh University

June, 1985

This thesis is accepted and approved in partial fulfillment of the requirements for the degree of Master of Science.

July 29, 1985
Date

Samuel L. Gulick
Professor in Charge

Daniel J. Hillman
Chairman of Committee

Eric D. Thompson
Chairman of Department

ACKNOWLEDGEMENTS

I would like to thank Samuel Gulden, my advisor, for his comments and support. A significant portion of my abstractions originated from his colloquium presentation on semantics represented by Culik's linked forest manipulations, and my subsequent study of Culik's paper. Similarly, I also thank Tom Morrisette, for whose course on data structures I developed the animation of Tarjan's algorithms. His insights of both industrial and academic matters were significant. Furthermore, I acknowledge my former employers, Samuel R. Frankel, of Frankel Engineering Laboratories, Inc., and B. Thomas Burson, of AMP, Inc., since it was while I worked for them that I first realized the need for a generalized debugger/monitor package, and contemplated its requirements and the domains it would have to span.

Finally, I wish to dedicate this work

"Ad Magnum Gloriam Dei"

To the greater glory of God

+

TABLE OF CONTENTS

1. Abstract 1

2. Introduction 3

3. An Abstract Model of Animation of an Algorithm 8

The Structure of an Executing Algorithm 8

The Code Domain 12

The Data Domain 16

The Code-Data Combination 29

The System Domain 33

The Code-System and Data-System Combinations 41

The Problem Domain 41

Combinations of the Problem Domain 43

4. Design Criteria for an Animated Algorithm 45

Human Aspects 45

System Design Aspects 54

5. An Example Animation 62

6. The Future of Animation 70

7. Conclusions 72

8. Figures 73

9. References 111

10. Appendix A (Externals for Figure 7) 114

11. Appendix B (STREAMER) 116

12. Appendix C (Externals for Figure 11) 117

13. Biography of the Author 119

TABLE OF FIGURES

Figure 1: "Bubblesort" original code 73

Figure 2: "Bubblesort", augmented to animate code domain 74

Figure 3: Low-order code animation 75

Figure 4: A simple animation of the data domain 76

Figure 5: A more complex data domain animation 77

Figure 6: stack control routine animations 78

Figure 7: animation of stack routines: 79
CREATESTACK, PUSH, POP, KILLSTACK

Figure 8: the difficulties of code-to-data animation 83

Figure 9: Tarjan's code for Prim's algorithm 84

Figure 10: Tarjan's dheap control routines 85

Figure 11: Prim (animated) 87
Global declarations, main algorithm,
DHEAPGENXY, DRAWDHEAPNODE, DSIFTDOWN,
DSIFTUP, DDELETE, DDELETEMIN, DINSERT

Figure 12: Sample animation of the Prim algorithm 97
(14 frames)

The Animation of Algorithms: Theory and Practice

by: Peter J. Floriani

ABSTRACT

The concept of animation of an algorithm (a program in execution) is examined, both abstractly and practically. The three important uses of algorithm animation are discussed: a teaching device, a debugging tool, and an aid in algorithm analysis.

An executing program is described abstractly as existing in four domains: code, data, system and problem. Each is subdivided into various hierarchical levels, delimiting orders of structural complexity. The members of each domain are given a representation allowing their display as a form, altering in time as execution proceeds. The several combinations (pairs of domains) are treated similarly. Examples of simple animations of the domains are presented.

The design issues suggest pairs of alternative approaches to an animation: ad hoc, which hand-animates one selected program, or "environmental", which primarily

machine-animates any program; program development environment: a comprehensive tool for editing, compilation, testing and animation, or a stand-alone tool, which pre/post-processes a program extrinsic to other usage steps.

Design of the human interface is discussed: the user views representations of the four domains in windows, independently controlled areas of a color graphics terminal, and has a wide range of control over the animation.

An example animation is examined: Tarjan's dheap implementation of the Prim minimum spanning tree algorithm. This provides an example of the code, data, and problem domains, and readily shows the applicability of animation to teaching.

INTRODUCTION

To animate an algorithm is to give a portrayal of its implementation structures as they change in time, thereby mimicking the execution of that program. The animation process may use any one of the dynamic art forms, such as cartoons, movies, or even ballet. Some of these forms may give rise to unique insights into the nature of algorithms, as well as possibly providing interesting entertainment, but they do not allow for simple reproducibility, nor are they readily available to most workers in the computing sciences. A form which does satisfy these requirements is animated computer graphics. This paper will consider both the abstract and concrete aspects of algorithm animation. It will also discuss the factors of human interaction, algorithmic structure, and system design, as they are involved in both ad hoc and environment-like algorithm animation programs. An example of an animated algorithm will be presented: Tarjan's dheap implementation of Prim's minimum spanning tree algorithm. Comments will also be made regarding the future possibilities of algorithm animation, in its relation to the areas of programming environments, semantics, and design.

Why animate an algorithm? Considered as a matter of

pure art, it is enjoyable! Watching an animated display of the execution of an algorithm offers the same exhilaration as conducting the performance of a symphony, even though the listing of the program provides a complete view of the program, just as the score of music does for the symphony. The conductor (the user) not only hears the music (sees the output), he also sees it being generated (the execution of the 'program'). Even more, it offers intense satisfaction, since the user, like the conductor, has complete control over the speed and other parameters of execution. But the score is still available, and so is the listing - which means the user may partake of both the static and dynamic forms. This static/dynamic ability, therefore, leads to the first (and most important) practical use of algorithm animation: as a tool which aids in the understanding of the action of an algorithm.

The ability to provide a fully controllable view of the internal, dynamic structures of an algorithm is a teaching tool unsurpassable in computing. This is because it offers a view of moving internal mechanisms, which previously could only be seen as multiple diagrams in books or erased and re-erased chalkboards, or as images in the mind's eye. This use of the computer as a demonstration tool might be compared to the use of the piano in music classes.

Brown University has developed a system of algorithm animation, intended primarily for teaching, which has been in use since September, 1983. This system, known as BALSA (Brown ALgorithm Simulator and Animator), was described in the SIGGRAPH '84 Conference Proceedings [1]; this article serves as the primary reference, and was the seed crystal for this thesis. Indeed, the author's intent in devising the initial animation program was to provide a pedagogical tool for the presentation of Tarjan's minimum spanning tree algorithms [2]. The animation of two network problems from Tarjan's work will be discussed in this paper. Problems of this type are especially fascinating when animated - in fact, any problem involving a natural "graph", tree, or pointer structure has a visually greater impact when animated.

Another use of animation is in debugging. Bit for bit, a greater amount of data is presentable by graphical means than by simple alphanumeric display. One might paraphrase the old saying, "a picture is worth 1024 words, especially when it is a dynamic picture of program execution". The author has made use of this technique in the industrial setting: when he worked at Frankel Engineering Labs, he used the graphics capabilities of various terminals to aid in debugging the geometric intersection routines of a

numerical control package. Additionally, he explored a peculiar stack-like structure which was being considered for use in the expression evaluation section of another package. A simple animated display was devised, using only an alphanumeric screen and cursor addressing. By watching the changing values of the pointers and arrays, it was demonstrated that the structure was correctly managed by its control routines.

The study of algorithmic design may also be pursued by means of animation. The use of simple graphical representations - static icons and dynamic actions - may provide insight of higher order than that obtainable from complex, more detailed views. One glance at a color representation of breadth-first and depth-first searches is sufficient to demonstrate this useful ability. The distinctiveness of the two patterns is so dramatic, it can scarcely be compared with a printed list of nodes visited in the tree. The ability to produce colored animative representations of such structures can aid in recognition of similar forms, and may yield new ideas by intuitive variations of that graphical representation. Finally, the visual portrayal of the dynamic execution of an experimental algorithm can provide a greater insight into the internal mechanisms, as well as assisting in

determining program correctness.

Thus, it can be seen that animation of algorithms has three main uses: teaching, debugging, and algorithmic analysis. The last two uses require a unified development system or environment to gain the proper measure of algorithm independence, and to remove much "busy work" from the programmer/developer. As regards teaching, this would also be preferable, as can be seen in the case of BALSA. For the animation examples of this thesis, however, the simpler technique of augmenting the code of an existing program was used - what we have called the ad hoc technique, as opposed to the environmental technique. The difficulties encountered in applying this technique will be discussed, but primarily attention will be directed towards the more desireable environmental approach.

AN ABSTRACT MODEL OF ANIMATION OF AN ALGORITHM

Before we can animate an algorithm, we must determine the nature of an algorithm in execution, and specify the manner in which its components can be displayed as moving in time. First, we will outline the structure of an executing program, as it is (ostensibly) implemented to solve a problem. We propose an abstract model of each of the major components - the four "domains" of code, data, system and problem - and specify a method of their animation. The realization of the abstract animation model is then described; sample PASCAL implementations for certain animation techniques are presented. Finally, we show the important inter-relations of components, which we term the "combinations" of domains, considering them as aspects of a complete animation system.

The Structure of an Executing Algorithm

Programs, as they exist on most modern computers, may be considered as having components in three domains: code, data, and system. While it is well-known that the operating system is merely a program, and the Von Newman declaration of "code is data" has not yet been discarded, we here wish to distinguish the differences of function,

rather than of substance. "Code" has the function of instruction, causing certain objects to be manipulated in a predictable fashion. These objects are "data", which have the property that a relationship exists between each computer-stored datum and some abstract intellectual concept, e.g. a number or name. The third division, the "system", is an organizational entity which allows independent streams of code to execute (jobs), supplies shared data objects (files, and other structures), and provides a means of communicating data to the "outside world" through input and output subsystems.

Now, what we have been referring to as an "algorithm" is really a well-defined collection of components of all three forms, and is what is normally referred to as a "program". (We have chosen to use the term "algorithm" somewhat loosely, since there seems to be no way of animating an algorithm - a purely intellectual abstraction - unless it had some real implementation.) The execution of an algorithm is the system-directed performance of the code of a program, manipulating some selected data elements, usually specific to each separate execution. This execution has been modelled in many ways, notably (in the abstract sense) by Turing machines [3] and linked-forest manipulation systems [4] and (in the concrete sense) by

any currently running computer.

We will extend the code-data-system combination with a fourth dimension, which, to some extent, fills the gap between the abstract algorithm and its implementation. This dimension, which we term the "problem domain", shows how the component parts (sub-problems) of a problem are solved by the algorithm. Unfortunately, the problem domain can only be implemented (on current machines) within the other three domains. One may think of the problem domain as being a kind of "algorithm correctness" specification, since it requires program augmentation analogous to that done in studies of "program correctness". (This may hint at another use of algorithm animation!)

An overview of the these four dimensions of the execution of an algorithm may be obtained through the following interesting analogy. Consider the computer executing a program as a person putting a jigsaw puzzle together. The code includes operations like "test two pieces for fit", "join the two pieces" and "set aside new merged piece". The data is the numerous variant shapes of the pieces, which, at times, may be organized into larger or smaller sub-structures. The system is the human puzzle-solver, who directs the execution of

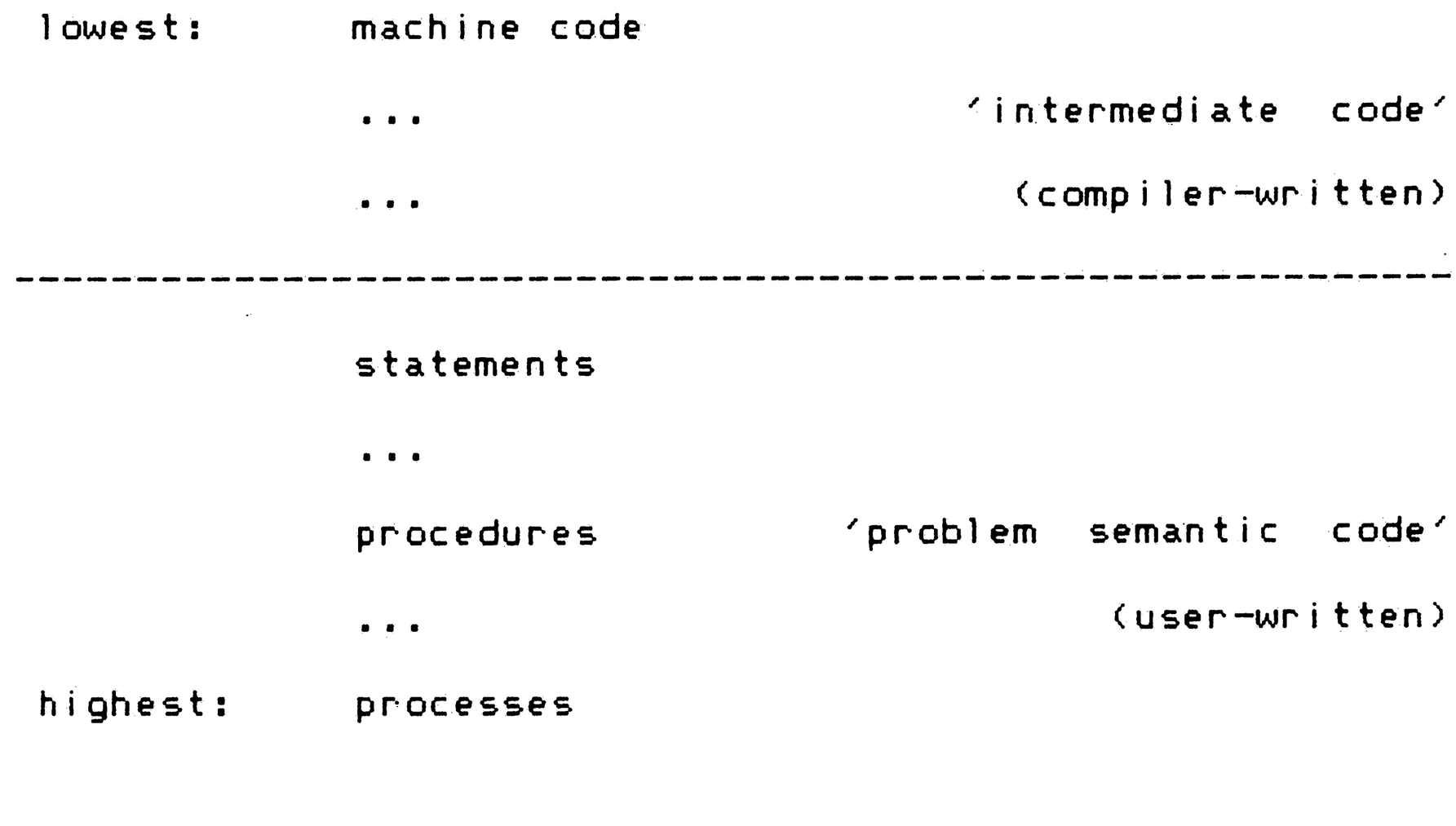
piece-fitting attempts, and the management of intermediate structures. If more than one person is solving the puzzle, we have a multi-processing system! The problem domain is the degree to which the jigsawed portions of the picture itself, as represented on the box, have been united. The sub-problems are, typically, the various dominant features of the picture (sky, buildings, etc.), since we usually solve jigsaws by "growing together" related pictorial pieces. A suitable animantion of puzzle-solving would be to film or video-tape the proceedings.

It is clear that the execution of an algorithm may be considered for observation in any of the four dimensions of code, data, system, and problem domains. Furthermore, each domain is resolvable into subsidiary units of measure - representing the various hierarchical levels which are distinguishable within that dimension. For example, in the code domain, one might distinguish the levels of Programs, Procedures, Statements, Substatements, and Machine code. These levels may reveal the intricate order of semantic complexity within each dimension. The usefulness of this decomposition (and the degree of its resolution) is purely subject to the user's desires, or the animator's intentions. Nevertheless, each domain should be

representable on (at least) one level of resolution.

The Code Domain

The code domain is a sequence of executed code in time: the instructions performed by a program, in the order specified by the problem currently being solved. This is a trace of the execution, not just a list of the code. The levels within the code dimension may be diagrammed:



Lying alongside this measure are secondary units, one of which may be termed "problem semantic code". These cover a portion of the user-written code area; they relate sequences of user code to semantic units solving some sub-problem, or step of a sub-problem (as opposed to the

possibly many lines of code which may be needed to express that step).

We will now specify an abstract representation of the code domain. Consider a log file, which is produced when an abstract machine modelling our real computer runs the program we are investiging. As it executes, each machine instruction executed is written to the file. This file contains one record per quantum of time (or machine cycle); each record is marked uniquely with the time of its creation. The record contains one field for each concurrent processor of the machine. In between machine cycles, records marked with non-quantized times are written to indicate initiation or completion of the higher structural entities of the program, such as statements, procedures, or the like. If we now attach some auxiliary mechanism to read this log file, displaying it in an appropriate form, we obtain a complete abstract model for an animated code domain.

This model allows us to readily specify the code domain portion of an animation system. In a compiler-oriented situation, a pre-processing step is performed before compilation: a procedure call is inserted between each serially executable code unit (i.e. statement). This

call passes a literal representation of the following line of code to a procedure which is part of the animation system. This procedure must perform the following tasks:

1. obtain a system clock timestamp
2. determine process/job id
3. format a record entry
4. log the record
5. display the record if requested

To guarantee proper synchronization when studying cuncurrent processes, the entire procedure should be executed in a (user) non-interruptable state. This procedure (known as 'AA', for 'Algorithm Animator', in the example of Figure 2) will also be supplied with a hierarchical level number, in order to allow the various levels of code to be discerned.

Unfortunately, typical languages have no other structural components between 'statement' and 'procedure' which may universally represent the 'problem semantic' components used in the program. A human intermediary is required to insert calls denoting problem semantics, since intuiting the steps of the sub-problems solved by a program from the code of the problem cannot be generally performed mechanically. Hence, animation of the problem semantic level can only be pursued by ad hoc techniques. One

simple approach, used in the BALSA system [1], is to insert these calls by hand when a selected program is prepared for animation study. In BALSA, the problem semantic steps are known as "interesting events", and they are displayed by a routine separate from that used for display of statements.

Figure 1 shows a sample PASCAL procedure "BUBBLESORT", which will be animated. An example of the augmentation required to animate the code domain for "BUBBLESORT" is shown in Figure 2. This method of hand-extending a program is straightforward, although there is also the risk of animating what the code means, rather than what it says. On the other hand, it allows maximum flexibility in clarifying meanings and subtleties of problem semantics, which is probably the only practical way of putting problem semantics into the code picture.

Some of the lines inserted in the example, notably those with second parameter '1', could have been generated by a pre-processor, since these merely 'pre-echo' the execution of each statement. An interesting special case occurs when a looping construct is animated: the loop statement should be displayed upon entry to the loop, and at each time the test for loop exit is made. The exit itself should also be displayed. Other "structured"

statements should similarly be "hailed and farewelled"; the animator might choose some of these to have distinct hierarchical levels.

To achieve animation of the code domain at levels lower than the programmer-written codes, an interpretive approach must be used. This interpretor, however, is unusual since it would interpret the compiled code of the program, which itself has already been augmented. The object code would also be augmented by the compiler to include tracing of semantic constructs as they are executed. As a very small example of this, see Figure 3. This is quite tedious to do by hand, although a compiler could readily generate code of this nature. Of course, any convenient "stack machine" or "three-address" operations may be used at the atomic level, rather than real opcodes.

The Data Domain

If one were to wire each bit of a computer's memory to a pixel of a graphics terminal, and then make a movie of the changing patterns as a program executed (where the frame rate of the film was the same as the machine cycle), the data domain of that program would be animated. This technique, although rather bizarre, is actually a logical

realization of the animation concept applied to data. (In fact, we would think that the idea would have its uses in design of new machines, or operating systems.) To a programmer, however, this method is unwieldy, moreover, though it provides a display of _every_ bit of data, it does not supply information easy to relate to the program being studied, since the information presented is of the lowest level.

Programs, even the lowest-level "bit-pushing" routines, typically make use of variables: the grouping of bits into larger structures. Programmers are mostly unconcerned with the implementation of variables as groups of bits (their micro-structure), but the organization of simpler variables into more complex ones (their macro-structure) is important. Our animation scheme must recognize these distinctions of the hierarchy of data structure by providing the ability to view the various levels of data.

We might organize data structures according to the following plan, useful for describing the (program) external data structures of an application system:

lowest: bits (each either 0 or 1)

simple variables (several bits)

records (several variables)

files (several records)

highest: data bases (several files)

Unfortunately, stacks, queues and other common structures do not neatly fit this plan. One can readily see the organization of bits into simple variables, and then the grouping together of simple variables, but the manner of describing all possible groupings is not apparent. Still, we do not wish to implement the animation of each different data structure uniquely, therefore we require a common representation for all data structures.

We have chosen to represent the data domain by the directed graphs of network theory, where the vertices are the elementary data items, and the edges are the relations which group the elements into structures. This may appear somewhat unnatural - however, not only does it provide a description of the entire hierarchy of data structures, it directly leads to an abstract model of their animation.

The data domain is modelled by a logging structure, each record of which is the forest of all the graphs of data structures used by the program. At each machine

cycle, a new record is written to this log. It is important to note that not only the elementary data items, but the graphs themselves, may alter during the execution of the program. We may superimpose graphs of various "colors" on the actual graphs written to this file, indicating structural relationships of higher orders, such as that associating all variables of a particular recursive level of a procedure, etc.

This representation leads to a straightforward animation of the data domain. At any point in time, we merely display the set of graphs of related variables in the following way: Partition the variables into super-graphs, such that each super-graph is closed under every relation associating any variable within it. Compute the coordinates of the vertices of a regular n-sided polygon for each super-graph, where n is the cardinality of the super-graph, located such that no two such polygons overlap. Each vertex of the polygon is marked with the name and value of the variable. Then, having associated a unique color with each different relation, vectors (line segments with arrowheads) of the proper color are drawn joining the vertices of variables which are in some relation with each other. Thus, both the changing values and relations of variables are observable with this method.

It is quite difficult to treat the data domain independently of the code in a real program. We will not do so, since our attempts always seemed to start with the "a pixel for every bit" technique. Instead, by assuming that the code-directed manipulations of data are a realization of the semantics of the attribute grammar associated with the code, we can represent the semantics specified by code augmentations. We will not give a theoretical justification of this statement, nor will we pursue specification of the complete set of code-augmenting operations required to represent general data-manipulating semantics. Rather, we will give a subset of operations, and show simple versions of their use.

The following operations will be used to represent manipulations of the data domain within the code domain:

rvalue(NAME) returns the r-value (reads the value) of the variable "NAME"

lvalue(NAME,value) sets the l-value (writes the value) of the variable "NAME" to "value"

create(NAME,type) creates a variable "NAME" of type "type", having no associated value

destroy(NAME) wipes out the variable "NAME", removing all associations with other variables

relate(A,B,R) relate variable A to B in relation R

rsons(NAME,R) returns the (possibly null) set of variables which are sons of NAME under relation R (that is, the set of A for which 'relate(NAME,A,R)' has occurred)

rfathers(NAME,R) returns the (possibly null) set of variables which are fathers of NAME under relation R (that is, the set of A for which 'relate(A,NAME,R)' has occurred)

We use sequences of these operations, together with a set of well-defined functions mapping any desired n-tuple of values into another (e.g. addition, multiplication, etc.). These sequences are inserted in the code of the program. Each augmentation will describe the changes made in the data domain by the adjacent (original) line of code.

These sequences are <u>not</u> actual code, that is, instructions! They specify the various inter-relations of variables (and their values) which change from instruction to instruction - representing the characteristics of the data domain which have been altered by execution of a code fragment. Only the sequences necessary to show the changed portion of data are actually inserted; these sequences then give an immediate realization of the animation of the data domain. As a very simple example of these sequences, see

Figure 4.

It is apparent that simple variables and assignment statements can be handled by this approach. Complications arise when we consider a structured variable, such as an array. Consider the program of Figure 5. We say that two variables of the same array have relation "SAMEARRAY" to each other. We also define the INDEXES relation to be the relation between the index (control variable) of the FOR loop and the array element to which it currently refers.

Note that we have added the following support routine to allow replacement of the currently defined son or father of a one-to-one relation, such as INDEXES.

replacerelation(A,B,C,R) deletes any existing relation R from A to some subscript of a variable B, and then adds relation R from A to C.

This is a simpler notation for the following pseudocode:

```
FOR $:=B.LOWESTSUBSCRIPT TO B.HIGHESTSUBSCRIPT DO
    IF B[$] IN rsons(A,R) AND A=rfather(B[$],R) THEN
        relate(A,B[$],NULL);
relate(A,C,R);
```

where $ is an unbound variable.

This can quickly get very difficult to follow, especially when animating several complex data structures in a set of procedures which may pass them both by value and by reference. This level of detail must be relegated to a preprocessor, or the compiler/interpretor, hence it is not suitable for ad hoc animations. We have not attempted any low-order animations in our experiments, rather, we have devoted time to the realistic portrayal of a few important non-trivial data structures, using what we term the "natural" or common representations. This leads to the topic of "problem semantic data structures", similar to the "problem semantic code" (which we referred to in the code domain discussion). This is, perhaps, the most important topic in the animation of the data domain.

Most of the major data structures in use today have representations which give a good "conceptual fit" - the pictures typically used in books [see, for example, 5] and explanations provide a good match between the structure (and its management operations) and that structure's mental image. Of course, the visualizations of the mind are not communicable as such, unless they are expressed physically, and, like the "chicken and the egg", we cannot say whether the concept, or the picture used to express that concept, has priority. While the philosophical aspects may be open

for discussion, we will take for granted the correlation of these common or "natural" representations with their abstract equivalents.

The conventions of "structured programming" and code sharability have led to two reliable features associated with most common data structures. A structure is nearly always referenced only through a limited set of control routines. In fact, some languages (or their run-time support libraries) provide these routines for certain predefined structures, but one data structure almost invariably implemented in this way is the file. Suppose, for example, the typical file is a language-related (i.e. compiled or interpreted) feature, rather than one supplied by the system. We make this supposition since, although we will later treat files as members of the system domain, they are properly a data structure, and our entire discussion of data structures is applicable to them. For a file, the set of control routines may include the well-known operations "open", "read", "write", "rewind" and "close". Often the implementation uses security constraints to explicitly prevent <u>any</u> access to files, <u>except</u> through these routines. While not all data structures may have their access paths limited in such a way, the program's use of the structure should respect the

structure's "privacy", and manipulate the structure only through the routines provided. Of course, data-flow analysis can resolve any exceptions from this rule.

The second useful quality of common data structures is that they have a well-defined internal form. This regularity does not necessarily imply the notion of homogeneity, such as the typical FORTRAN array. We refer to a "typed" structure in the sense of PASCAL, or a schema-defined construct, such as some data bases [6]. This "type" or schema must be available at some level of machine-processability, whether it is at compile-time or run-time, and variables of this type remain of this type, unless changed through regular means. (Irregular means would be use of PEEK and POKE in BASIC [7], or RPLACA and RPLACD in LISP [8], for example.) Again using the instance of files, most systems allow specification of a file's record size or "type" (text, binary, etc.) only at the time the file is created or "open"-ed. Typically, no facility is provided for conversion of the file from one form into another (e.g. text to binary) in situ.

Although we know that we may always choose the "polygonal graph" visualization of a data structure, with animation by augmentations mimicking the attribute

semantics, we will resort to it only for low-level or extreme cases, since it will rarely produce the common representation of many important structures. We will avoid this difficulty by choosing to animate them on a higher level - that of the structure's access and control routines. This means that we must rely on the regular definition of each data structure, and controlled access to it through a small set of control routines. As we have seen, both of these requirements are met for most frequently occurring data structures. For them, animation is straightforward, since the regular structure results in a machine-performable representation, and the control routines delimit the code for the actual animation.

A common example is a stack. Let us define the members of the stack to be integers, and provide the following four control routines:

CREATESTACK(name) creates a stack with name 'name'

PUSH(name,value) pushes 'value' onto stack 'name'

POP(name,var) pops top of stack 'name' into 'var'

KILLSTACK(name) destroys stack 'name'

It is easy to create animated versions of each of these routines, without considering the details of the stack's implementation. Figure 6 shows sample graphical representations of these operations, using the common form

of a stack. This figure contains four separate 'frames' of a code sequence, starting just at the creation of the stack. Note that these frames are "artist's renditions", not outputs from any program. A 'standard' heading region in the upper left corner has been added, providing both the actual source code and the time of its execution, in order to give a feel for the time flow involved in the sequence. It should be clear that the KILLSTACK routine merely deallocates the display structure acquired by the CREATESTACK routine, or visually marks it "no longer active" in some way - we have not shown this action.

We give the code for the animated 'front-ends' of the above routines in Figure 7. The four original routines have been renamed with a prime to represent their new internal existance. Remember that we are not concerned with the details of their implementation, which we do not provide! We use the (unprimed) names as the names of the animated front-ends, in order to avoid changing every call to these routines throughout the program. Note that we have disregarded some complicating issues in these implementations, such as what to do when the stack grows bigger than the screen area. This and similar problems of structures larger than a window's area will require a control device such as "scrolling", which will be

discussed later, but it is clear that their resolution involves changes only at these locations.

A similar, but somewhat more complex technique for data structure display is used in INCENSE [9]. This is a run-time system supporting the Mesa language on the Xerox Alto personal computer. Relying on tables generated by the compiler, it requires no source code modification. Upon arrival at a breakpoint during execution, it provides the user with the ability to graphically display any Mesa variable. While INCENSE is intended solely for debugging use, the approach used for data display is worth of note. Each data type has an associated collection of procedures and data, termed an Artist, which produces the display of a variable in any manner specified by a previously defined Format or Layout. A Format may, for example, specify information to be displayed in either an "internal" form (as variables) or as some analogous form (such as a thermometer, clock, or other device), whereas Layouts handle placements of pointer-constructed forms. INCENSE is highly user-oriented in presentation, and the user may interactively select where a variable is to be displayed. While we feel that it is an important variation in techniques of data domain animation, we do not have room to pursue this approach further.

The Code-Data Combination

We now turn to the first of the combination issues, certainly the most complex, as well as the most important: the inter-related animation of code and data. This topic may be considered to exist in two directions: code-to-data and data-to-code. The code-to-data direction provides information on variables in the context of the source code (an extension of the animated code domain), while the data-to-code direction shows the origin in the source code of changes made to the data structure (an extension of the animated data domain). The first is not easy to implement, although it is useful in all purposes of animation; the second is very difficult and convoluted, but it offers great advantages for debugging and algorithm research.

When code is animated as previously discussed, the statement (or language fragment) is seen as it originally appeared in the source, except for possible prettyprint reformatting. Often we wish to see what values are currently associated with the variables as they appear in the code. This is termed the code-to-data direction: see, for example, the code in the second frame of Figure 6:

```
PUSH('ABC',123);
```

Here, rather than show the original call, we have shown the values which were passed to the "PUSH" routine. Although

the first parameter of PUSH is a VAR parameter, it is read-only throughout that routine, being passed as VAR for efficiency, so we may show it as a value.

From the example mentioned this might appear to be a simple task - merely show the value of each variable as it appears in the code - but there is more to be considered. First, not all values are representable "in-line" in the code. Arrays and other structured types, not to mention files, must be notated in some way. Another problem in some languages is induced by the read-only nature of pass-by-value parameters versus the possible "hidden assignment operator" of pass-by-reference parameters: we face a dilemma in notating the parameters of procedure calls. We attempt a solution in the code of the fourth frame of Figure 6 by insertion of the assignment operator :

```
POP('ABC',:=456);
```

A third difficulty arises since the original source should not be ignored - we should see the variables as they were coded, as well as their values. This is complicated when the variable has a non-trivial scope, i.e. formal parameters. In Figure 8, the actual parameters of "distance" called within "distance2arc" may be displayed relative to "distance2arc" or the main program. We suggest that the approach requires the use of layered screens of

source code, with variable values interspersed, and a wide-spread use of color (and probably a strong reliance on interactive selections of views), but we will leave the details to be specified in future work.

The direction of data-to-code is very complex, both to describe or to implement. We desire to record the exact statement (or fragment) which causes a change in data structure, along with other pertinent information. Why bother with this? Primarily, because one of the most difficult questions to answer in debugging a program is "How did that variable get to be that value?" - hence our interest in this difficult issue.

An instance of the data-to-code relation occurring in the industry is the "log" or "audit trail" concept used in data base backup and other applications [6], where each alteration made to the data base is logged to a file, recording the "before" and "after" states of the data, time of action, and program-identifying information. This is, of course, only a simple subset of the data-to-code relation.

Here again, difficulties arise in attempting to give a general means of implementation. Occasionally the true

cause of a variable acquiring a particular value is not directly related to a particular assignment statement (witness the pass-by-reference issue mentioned above). Many times the values of other variables (involved in conditional or looping statements) are the indirect, yet more informative, causes of a particular value assignment. Another problem is related to the indistinguishability of nested recursive routines. Of course, each new level of nesting may be tagged with a depth number, but a clean representation is still needed. The final issue is probably insurmountable in any (current) practical sense: given that the direct cause of value assignment is available, show the antecedent causes back to the start of the program. Yet, one should refrain from questioning the utility of this issue until one actually does some debugging!

As in our discussion of the code-to-data relation, again we can only give some suggestions for further research. If the location in the source code of each assignment of a variable is recorded, a partial solution may be obtained. Perhaps a pointer to the code domain's atomic log should be recorded at each assignment, and procedures supplied which display information from that log. Despite the obvious desirability of a data-to-code

display, its complexity prevents us from going into further detail.

The System Domain

The operating system is a program - therefore it is composed of both code and data. The distinguishing feature of the operating system, setting it apart from other programs, is its ability to perform input and output to external devices, and its control over (real or apparent) concurrency of multiple sequences of code execution. Since each operating system has its own unique entities and control methodology, it would be unreasonable to include the many diverse members of the system domain in one general abstract description. However, both device input/output and concurrency can be described within the code and data domains already defined. Thus, we could resort to examining the system as a program, applying the techniques already devised for both code and data. An analysis of the system as a program would be valuable for its own design and debugging, but, as the object under examination is usually a program running within the system, the inner details of the system's operation are not pertinent. Therefore, we will view the operating system, not as a program itself, but as code and data components, used within, but separate from, a particular

program.

On today's mainframes, one often finds many users, each engaged in solving problems independently, and others cooperating in solving problems. Our general treatment of animation is on a "problem" level: there exists one main program directing solution of a problem. This program may cause execution of new dependent or independent programs, all committed to partial solution of the master problem (as controlled by the main program). Of course, in a typical data processing package, the many functions of data-entry, updating, report generation, and even data backup occur simultaneously yet independently - but each independent program may be considered to contain portions of the main program which direct the order of access to the shared data uniting the many programs as sub-solvers of the master problem. Thus, in a sense, the main program is the operating system itself, as it is responsible for the concurrent access to shared storage, and the scheduling of simultaneous programs. So the animation of the system domain is really a monitoring of the operating system itself. This animation, however, must be distinguished from performance measurement tools (which measure CPU and memory usage in various ways) and other system management tools (which keep track of usage by users, accounts, or

other managerial structures). Animation shows different levels of detail than either of these system monitors.

In order to animate the system domain, we will make several assumptions about the interface between a program and the system, and we limit our analysis to those common abilities provided by a typical operating system. These limits are placed so as to avoid the difficulties resulting from the varying representation of similar forms on different computers; we believe that we have included the major capabilities available on many systems. Our specifications should be readily transformable into those for any system with which the reader may be acquainted. Any system feature tied to some linguistic construct, e.g. ON OVERFLOW and other interrupt-related devices, can usually be transformed to meet these specifications.

Regardless of the usual language constructs which implement the abilities provided by the operating system, we require that all requests and transfers of information must occur only through a procedure call (or related language construct). This "proceduralizing" is done to regularize the syntactic occurances of system requests, giving a common structure to each preparatory to animation. Note that we distinguish the facilities provided by the

operating system from those of the run-time support library: the code for such things as trigonometric functions, type conversion, and formatting could be performed by code within the program; such routines are placed in libraries primarily for convenience, rather than functional uniqueness.

We will only investigate animation of the following important system entities:

Data forms:

1. files - a data structure managed by the system, usually of the form "ARRAY OF type", accessible from more than one program, but preserving the integrity of each component or "record".
2. mailboxes - data structures of any form which are accessible from more than one program, hence requiring certain integrity control not needed by variables private to a program.
3. devices - a special form of a file, in most cases only accessible sequentially, and for which a certain time must elapse until the next record may be read or written.

Code-in-execution forms:

1. programs - the synthesis of code-data-system domains in a state of executability

2. processes -a program which executes simultaneously with all others, related in a tree fashion to some others, the root is called the "main program".
3. jobs - a main program (and possible descendents) called into existance by some user action (or by code within a program)

We will not examine other less common system abilities.

Having specified the entities controllable by the system, and examined the restriction of those controlling actions into procedures, we can proceed to specify the animation of the system domain. First, a procedure call to each different system function is enclosed in a "front-end" routine. Each of these routines is then augmented by animating code, designed to portray the code form or data form being controlled. The program under study is then altered to call these "front-end" animations of the system, instead of their originals, either by ad hoc alterations, or environmental mediation of the calls. Although we need a separate animating routine for each different component, we only have to produce this code once to animate the system domain for any programs running on that system. We are free to choose either the environmental or ad hoc approaches in which these animation are performed, but the concept of sharability of code would indicate some form of

system animation environment, even if the other domains are animated in an ad hoc fashion.

All of the system's features, including the execution of independent jobs, are considered as they aid in the solution of one specific problem (as requested usually, but not necessarily, by one particular user). That is, the internal detail of jobs created from a user's main program, their descendents, and files accessed by them, must all be available to fully animate that main program. However, details of jobs not belonging to the user may often be inaccessible, due to constraints of system security. A complete animation implementation requires at least functional indications to be available, so each job step can be seen as it incrementally solves the user's problem. During the author's employment at AMP, he wrote a program "STREAMER", which achieved this within the existing security framework [10, and appendix]. Since program animation in the industrial setting will almost invariably be used for debugging, and done by system programmers rather than end users, we feel that its use will not threaten security. In any case, the ideal (interpretive environment) approach can always be used to provide full animation of system details, independent of system conventions. For some operating systems, this may be the

only possible approach, as it may be too difficult to construct front ends for many or all system actions.

The previous descriptions of the code and data domains are useful in animating portions of the system domain. The code forms belonging to the system domain may be resolved into a continuation of the upper hierarchical levels: after "program" (equivalent to "process") comes "process-cluster", then "job" then "job-cluster", then (the universal level) "main-code". Similarly, the data forms may be extended upwards, but since they mainly lie alongside the data domain already considered, we only state the extensions: after "file" comes "data base" (in the sense of a collection of files), then "data system" then (the universal level) "main-data". The executable union of the main-code and main-data forms the entire problem-solving structure within the computer.

Animation of the system code forms is more closely tied to their levels than the remainder of the code domain, because of the partitions commonly used by the system: the main-code is necessarily a collection of job-clusters, each job-cluster is a collection of jobs, and so forth down to processes (usually the atomic code form in the system domain). As before, we will use a log file (with a record

for each concurrent processor) to animate the code domain. It is also useful to provide a "cutaway" view of this extended code domain, showing the formation of larger components from smaller ones, tagged with the system's identifier at each discrete level (job, process, etc.).

The situation is somewhat simpler for the data forms, as no new data structures are introduced (the distinctions are of a secondary, i.e. problem-semantic, nature). In fact, the constraint of file access to the file control routines clearly gives the attribute-semantic view of the "file" data structure previously referred to. The major difficulty in the animation of system data forms is achieving good representations of the gigantic size of the typical file.

Since the algorithm we animated did not require system activity, we have not explored the system domain as deeply as the others, nor have we developed any examples. However, the "front-end" method of animating the system domain is consistent with both environmental and ad hoc animations, and with our developments of the code and data domains. Therefore, we will leave further details to future work. We feel that the ultimate system animation will involve the environmental approach, and allow

selection of interpretors simulating various common operating systems, permitting analysis especially of concurrent programming, and programmatic invokation of system-supplied utilities such as data bases.

The Code-System and Data-System Combinations

Since our discussion of the system domain showed its mapping into portions of the code and data domains, our comments on the code-data combination apply equally to the code-system and data-system combinations. Our mentioning of the "audit trail" logging for data base recovery can now be seen to really be an instance of the system-code direction. It should be clear that the information documenting these relations between code or data and the. system must be tracked within the hierarchy of the code domain, as extended by the system.

The Problem Domain

The last of the four domains to be considered is that in which the user's problem is solved. Recall that we are satisfying the request: "Show me how much of the problem has been solved". Mechanical solution of this would probably require a formalism of human problem-solving not

currently available. Fortunately, since problems solved on computers have a mathematical nature, the task is somewhat simplified. We will tentatively model the problem domain by a "forest" of graphs. Let the nodes represent the "atomic" level of the problem. The edges link the nodes (or subgraphs), as they are related by the algorithm. Other graphs are superimposed, each of different color, which indicate sequences of possible "states" of the solution. The program being studied is then augmented with procedure calls, manipulating this forest as the program executes. While this short exposition is hardly a general-purpose animation, it may suggest approaches for various problems. Initially, we can see that it is properly an ad hoc animation, since it requires assignment of the problem's graph-manipulating procedure calls to specific points of the program code. This follows because, as far as we can tell, only the designer of the algorithm can define its meaningful sub-problems.

The problem domain, then, can only be animated by the ad hoc approach. The animator must consider which, if any, of the code or data structures within the program will provide a good representation of the problem being solved. If the reader considers a few of the algorithms usually studied in an algorithms course, the technique becomes

apparent:

Sorting: show the array being sorted

Searching: show an "eye" looking for the desired item in its structure, and put a check mark at mismatches

Minimum spanning tree: depending on the algorithm, show the spanning tree, or partially considered subtrees.

Numerical integration: show the curve, the regions of approximation, and their areas.

Parsing: show the input tape, and the parse tree.

These suggestions usually take the form of a data structure, either already existing, or constructed in order to model the problem domain. One is reminded of the pictures used in cookbooks for cake decorating, showing the step-by-step process forming the final product. As an alternative, one might use a checklist, marking off completion of each step - this is then an extension from the code domain.

Combinations of the Problem Domain

Without even a partial description, it is virtually impossible to give any details of the various combinations of the code, data, or system domains with the problem domain. The general idea would be to show the relations between a sub-problem's solution and the corresponding

segments of the other domains. If the problem domain is partially mapped into a data structure, or code structure, these may be implementable. We feel that a case must first be made for the utility of the combination displays, though completeness insists on their availability, and aspects of algorithm analysis may be simplified through their use.

DESIGN CRITERIA FOR AN ANIMATED ALGORITHM

We will consider three aspects of design: the human interface, coding, and system support. Our comments are aimed towards the design of a environmental animation system, but they are also useful for *ad hoc* animations. We will not specify any actual design, the interested reader is referred to [1] for details of BALSA.

Human Aspects

In planning the human interface of an animation system, there are three major categories to be examined: what the user sees, what the user does, and other additional user requirements.

What a user sees

The user will be provided with a color graphics terminal of sufficient resolution to allow small characters to be clearly read, even when written on an angle. The screen's programmable area will be subdivided into "windows" or viewports. A window may, during the course of the animation, contain a representation of any of the four domains code, data, system, or problem. One window should

always display the current program, time, version level, and other pertinent global information. Another should show a master log of executions, edits, etc. possibly in the style of the DAYFILE [11] of the Control Data Corporation's SCOPE operating system.

The contents of a window are diverse, as each hierarchical level of the four domains may use of several different methods of display. Typically, those showing the "log" type information will show the current log record, and a limited number of immediately past records, the older ones having "scrolled" off the top of the window. Others may allocate areas for variables, represented in their usual forms, or as graphs; still others may use analog representations of real-world objects in motion, through manipulation of icons, or by alteration of graphics of indeterminate structure. Windows may overlay each other; this occurs when a "zoom" to show finer detail has been requested, or, as in BALSA, to represent recursion. Color will often be used to distinguish among the different relationships existing between items in a window. Usually, graphical data will be confined to the windows, except possibly to implement the various combinations of domains, in which case arrows may be drawn to represent the link between the code domain and the data domain.

What a user does

The user of an animation system has several facilities for controlling the display. Most graphics terminals provide a wide range of different input devices, but we will only consider the "logical" aspects of user input. We divide these into the categories of display control, execution control, edit/compilation, and utility functions. In keeping with the "user-friendly" nature of the concept of animation, a good help facility is also important. When the system is used for instruction, those functions which allow modification of the algorithm, or of run-time structure, may be disabled.

The user controls the display by selecting one of the following functions:

1. Define the physical area of the screen to be used for a particular window
2. Select the domain, and level within it, to be displayed in a particular window
3. Select or alter the colors used, globally, or within a window
4. Cause zooming (magnification or shrinking) in a particular view, either dynamically or statically, within the original window, or into another

5. Scrolling (scanning in a linear fashion) of incompletely displayable views such as logs, large arrays, files, etc.
6. Selection of "pretty-print" and other formatting parameters used for presentation of code (especially useful when the code-to-data combination is being animated)
7. Selection of various formats for display of data values: i.e. precision of real data, truncation and expansion of strings, and internal vs. external as well as analog vs. conceptual representations.

When an animation environment is to be implemented, other features of display control may become apparent.

The second set of controlling features to be examined are those which govern execution. Some of these are:

1. Begin (possibly concurrent) execution of a program, specifying its actual parameters, such as files, etc. An interesting aspect to be handled is provision of the load-time processing normally available in the system. System parameters (e.g. priority) are specified at this time. Windows must be allocated for the standard input and output files, if used.
2. Interrupt execution, causing a "pause" state in all

current programs being examined (unless otherwise disabled). Note that this function is performed during the execution of a program, rather than at a "command" level.

3. Kill an (interrupted) program, destroying all its execution structures, or removing access from permanent ones
4. Change the "state of executability" of a process (e.g. when one waits for another) - used in study of concurrent processes
5. Set up triggering of displays of certain items when a condition arises during run-time. (e.g. "show file ABC when it is opened by routine XYZ")
6. Provide various operating system commands useful in altering the run-time state of a program
7. (This is the wierd one:) Reverse the execution of a program - showing the various domain representations as they change backwards in time. This, in general, requires an additional set of internal definitons when a program is animated, and may not be useful in every instance.
8. In advanced systems, allow "immediate" execution of various statements. This may be only be truely practical in languages such as LISP (see [12] for one approach)

The above options are important capabilities for a problem-solving algorithm animation environment. Extended forms of control will be needed to enable support of large simulations of complex concurrent manipulations.

Whether the animation environment is used for debugging, algorithm study, or preparation of teaching aids, the program under investigation must be able to be edited, and subsequently (re-)compiled. As augmentations at pre- (or post-) processing time will undoubtedly be required, these abilities are best handled within the environmental scope. We recommend the following abilities, some of which we have selected from the PECAN system [13], the program development tool which accompanies BALSA.

1. Editing and compilation must be available within the environment. Associated pre/post-processing should be invisible, as well as any "front-end" or other run-time mechanisms.
2. [PECAN] Use of incremental semantics, allowing compilation as editing is performed
3. [PECAN] Feedback of syntax/semantics violations at edit time
4. [PECAN] Undo-able editing
5. Directory and archival techniques to allow keeping of several variant versions, also to mediate multiple

programmer development of large programs.

6. Complete cross-referencing of all data domain items.
7. "Zoomable" code: the ability to window out the declaration of a procedure from an instance of its calling, and similar for data structures.
8. (For advanced systems) "zoomable" compilation, to show the production of descending internal levels of code

Many of these abilities are already found in various current program development environments, the proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (which includes [13]) has much further details of their aspects.

The last category of user control options, the utility features, are all those which are useful to either development of animations, or their study. Many obvious commands, such as "leave the environment", we have not included. Rather, we only mention a few unique to the animation system itself:

1. Request a "display dump" of the current screen. This may be interfaced to a plotter, to produce a publication-quality diagram.
2. Request logging of a particular view to a file, for subsequent analysis, or to produce a transcript. (The

following example of animation was produced in this way, although it was an ad hoc animation, as well as an ad hoc graphics interface.

3. A combination of the above, which, when linked with appropriate hardware, produces a complete movie or video recording.
4. The (interrupted) assemblage of program and its run-time structures may be "saved" and recovered at a later time.

The help system should be "context-sensitive" - it should be able to identify the site of its invocation, and present the material most useful for that site. It should use a dynamic window, overlaying a portion of others, which is later restored when the help facility is exited. The animators and other code-producing users of the system should be encouraged to implement "help" within their work, by making available run-time structures of the same form for use in the programs actually being animated. This same structure may serve the purposes of animating the various "problem"-related domains, and also encourage more thorough documentation.

Other additional user requirements

As we have already covered the "output" (what the user sees) and the "input" (what the user does) of the animation system, the remainder of the user requirements is primarily exterenl to the computer (or program). Almost certainly, a textbook of some form will be provided, detailing other aspects of the algorithms being studied. Of course, sets of animations may be embedded in complete tutorial programs, allowing study of algorithms as code, as concept, and aspects of their history and efficiency. Whichever extension, paper or video, is used, however, one or the other will be necessary for academic usage. As regards the artistic aspects of animation, this amounts to the provision of a "program" (in the non-computing sense of the term) for the event about to be enacted, as well as a "libretto and score", allowing a full appreciation of the work.

SYSTEM DESIGN ASPECTS

We will now discuss some of the issues which determine the actual implementation of an animation system. These topics are considered on a gross level, in terms of the desired product, rather than addressing more subtle points of programming, which we consider in the next section. The overall design of a program for the animation of an algorithm will require decisions on the following issues:

ad hoc versus environmental

program development environment vs. stand-alone tool

single-user versus multi-user

single or multiple language

the levels of detail for the four domains

modes of user control over execution and animation

the problem domain animation method

system domain:front-end "native" system or interpretive

support software in the operating system

hardware (graphics terminals)

We will make a few suggestions on each, hopefully enlightening them to prospective animators.

We have used the term "ad hoc animation technique" to refer to the animation of exactly one program (or a extremely limited selection), distinct from the

"environment animation technique" which can (usually only partially) animate any program submitted for processing. An ad hoc animation typically takes a tested, "mature" program (or one of reasonable size, as with our example) and accomplishes the animation strictly through hand augmentation. This is both a plus and a minus: plus, because animation of any problem domain desired is achieveable, (as we have stated before, we foresee no practical machine animation of the problem domain), and all kinds of clever analog cartoons can be obtained; minus, because no algorithmic augmentation of code (which is vastly easier) or data (which, because of data-flow analysis, is vastly safer) is done (except, of course, in the case of a hybrid approach, or the additional use of special pre/post-processors). An environment system should be able to accept any executing, but not necessarily debugged program (since one motive of animation is debugging) and, through pre/post processing, possibly combined with, or replaced by, animative interpretation, produce displays as desired with little to no programmer effort. The rating of environments is more or less opposite to the ad hoc approach. Hence, unless there is a compelling reason otherwise, we recommend a hybrid approach, but centered on the environmental technique. There should be pre/post processing, coupled if required

with an interpretive scheme, and also facilities for easy insertion of problem-domain specific animations.

From our viewpoint as system programmers, further enhanced by our having one foot in industry and the other in the academic world, probably the single neatest idea we have encountered recently is the "program development environment" concept, "PDE" for short. The Proceedings in which [13] is contained, for example, show that others feel that way also. While we feel that a closure of the required properties of the PDE is still in the future, it difficult to imagine excluding animation. Obviously, we are strongly biased towards the "PDE" approach, as opposed to the stand-alone tool. However, it can be seen from the journal of [13] that design, not to mention implementation, of a good PDE is a lengthy process. Therefore, we would expect that the stand-alone approach will be more common, until PDE's become more widely available. Briefly, as regards animation, virtually every plus is on the side of the PDE approach except the lengthiness of development and implementation, and their comparative scarcity.

Strictly speaking, the issue of producing a single user animation as opposed to a multiple user animation system relates to the exclusive or concurrent manipulation of

portions of the program being studied. We mean concurrent manipulations to be distinguished from the multiple instances of animated programs, which may be on the same machine, or on a network of machines (as in BALSA). This question is predicated on the use of a program development environment approach to the animation. We suggest that if possible, multi-user abilities be included in environments, though they may not be required by most applications.

Animation of the system domain has two alternatives: the "native" approach, in which every reference to an operating system feature is front-ended with an animatable routine, or the interpretive approach, which simulates either only the system domain, or the entire program. These can get into the most difficult implementation areas, and the decision is not independent of other design issues. The ad hoc approach would virtually preclude the interpretive system domain, unless tools for this already existed within the run-time facilities. (This is like expecting system A to have some properties of system B, and programmatically available in the identical way: as if the program running under A were really running under B!) If an interpretor were to be built, it would therefore allow for "cross-execution", which naturally could be animated.

Hence, study of operating systems other than the "native" one could be accomplished. However, this can cause an animation system to become ridiculously complex. If one is using one particular system, we feel that it would be reasonable to have <u>one</u> program which enables animation of that native system (by whatever means); then, any <u>other</u> systems of interest are to have <u>their own</u> programs to allow cross-execution study. We do, however, believe that they should be unified whenever possible. Similar statements apply to the single/multi language question.

The selection of the levels of animations to be made available for each of the four dimensions is not easy. Again, the decision between the <u>ad hoc</u> and the environment approaches is the controlling factor. Unless the algorithm was extremely valuable, little would be gained by animations on many levels of, say, the code domain, if done on an <u>ad hoc</u> basis. An environment, however, can allow the choice much more readily. We suggest that an environment implement as many levels of display as possible, consistent with their ability to be performed by machine, and the result of the decision of the "native" versus interpretive system domain. Of course, an <u>ad hoc</u> program may animate <u>every</u> level. But we would strictly limit animation to <u>only</u> those levels of real

interest and utility.

The most perplexing design issue is the animation of the problem domain. One feels that there is some way of getting at least a partial mechanical animation of it from a given program, but we can offer no characterization of a feasible method. This would seem to require the use of the ad hoc technique, without exception, in the animation of the problem domain. Our comments on hybrids, nevertheless, are nowhere more to the point. Clever "hints" offered by the environment, and succinct forms of augmentation should make the machine-readable specification of the problem domain as natural as documentation, which, in a sense, it really is. It is also true that certain classes of problems may be delimited, then the "obvious" parts of those problems animated according to previously defined scripts. We feel that much exploration is required here, even though it borders on the philosophy of problem solution.

It seems that innumerable chances are offered in the computing field to prevent the proliferation of machines of diverging form. Nevertheless, it is difficult to find a common set of capabilities on hardware or software; yet while software is fluid, and, given certain elementary

abilities, anything can be accomplished. Not so with hardware - especially when graphics is considered. As a printing terminal can never erase or back up, nor can an alpha display draw circles, nor can a black-and-white screen show red, green and blue. We expect that a color graphics terminal of reasonable resolution is available, capable of communicating with the host at a sufficiently high speed to enable clean animation. The volume of data, however, will typically not be quite as large as that of required by CAD-CAM implementations. Furthermore, most 2-d or 3-d graphics effects will not be necessary. A "mouse" or other graphics input device may be convenient, as long as it is not the only input device (or main one!). Additionally, recall that much of the animation is textual, but not necessarily restricted to purely alpha-screen character positions, which are typically too large. We take this opportunity to suggest to the terminal industry that new screens be made larger. Until that happens, the letters used for animation will have to be made smaller.

One of the elementary requirements of the software which we require is the ability to create concurrent processes, and likewise cause interrupts between them. An important corollary is the ability to recognize user input while performing output on the same channel - akin to a

"user-abort" request, but trappable to any desired process or routine. We expect the common set of file manipulation routines, as well as all the usual system capabilities (or their equivalents). We strongly encourage the use of a device-independent system for graphics control such as the ACM SIGGRAPH CORE [14] to simplify window manipulation.

Any design must obviously take into account what the user sees, and what the user does, and how they can be achieved. We have already given a reasonable set of criteria for both of these; normal techniques of design for user display and control apply, except for the issues covered in the paragraphs on hardware, and system support. That is, the need of graphics "windows", and of user "interrupts".

AN EXAMPLE ANIMATION

As we mentioned in the introduction, this program was developed for use as a teaching aid - our intention was to animate the major algorithms from Tarjan's book, "Data Structures and Network Algorithms" [2], used in a course of the same name. Although we had completed only two of the minimum spanning tree algorithms, it was apparent that either of them covered all the major points of this paper (except the system domain, which did not appear in any of them). We chose Tarjan's dheap implementation of the Prim minimum spanning tree (MST) problem to use as an example of a complete animation.

For reference, we summarize the problem, and the properties of a dheap. Given an weighted undirected graph (a set of vertices, a set of pairs of vertices, and for each pair, a real number called the weight), find a spanning tree (an acyclic subgraph of the given graph containing all of the original vertices) such that the sum of weights of edges in that spanning tree is minimal. Prim's algorithm (Figure 9) requires selection of a "source" vertex which becomes the root of the tree. At the source, each edge incident to it is examined, and providing that edge's cost is smaller than the cost already found

between the source and the vertex at the far end of that edge, it is "colored blue": marked as being a possible member of the minimum spanning tree. Otherwise, it is colored red. When all incident edges have been examined which leace the source, the vertex at the minimum cost from the source is selected, and the process is repeated using that vertex, rather than the source. This is where the dheap structure is useful. A dheap is a tree in which each node has no more than d children, and such that the key value of each node is less than that of any descendents of the node. (There is also a dual using greater than.) Tarjan gives the algorithms (see Figure 10) for the various control routines for a dheap, implemented as an array, in what he terms "intrinsic form" (the structure does not use pointers).

As we felt that the key components of Tarjan's Prim MST to be understood were the dheap and the set of "blue" edges (those selected for possible inclusion in the MST), we concentrated on their animation, with enough of a tracing of the instruction sequence to see when they changed. Hence, we apportioned the screen into three areas: the first would represent the "blue" edges, the second to show the dheap, and the third to trace the dheap calls within the main routine of the algorithm. One should

consider the fourteen frames of Figure 12, as we now discuss the contents of these three windows. We have attempted to give a good portrayal of the operation of the animation in this lengthy figure, but, as motion in static art is most elusive, we have not succeeded. We suggest that the reader acquire an actual animation program to get the true effect.

The first window shows a polygonal figure, and some of its diagonals. This is a convenient technique we have used to display graphs - the vertices of the graph are mapped to those of the regular n-gon, with n the number of vertices, and labelled with the vertex identifier; then, the diagonals are added, corresponding to the edges, on which are placed arrowheads and weights. Except for the first frame, which shows the original graph, this representation shows the set of blue edges. These are the current candidates for membership in the final minimum spanning tree. Note that an edge which is blue (for example edge (1,2) of weight 8, in frame 2) may later be discarded - changed to red (changed in frame 9, thus not seen in frame 10). Red edges are not shown in the figures. It should be apparent that this image is an animation of the problem domain - the blue set is recorded, but the red set is not. A problem-related image (the n-gon) is used, requiring some

contrived structure not otherwise present in the algorithm - but it is quite effective!

In the second frame, we show the dheap, in the "natural" form used for dheaps: a "V"-shaped table of the nodes of the heap, with the root at the top (the smallest value), and branches extending downwards from each node to its children. We chose to use d=2 for our example. As two components are being stored in each node of the heap - the vertex and the (minimum so far observed) weight of an edge leaving it - we display both of them, the vertex above the weight. The weight is the value used in forming the heap order, which is readily observed, especially in frame 9. When one combines the code in the lower area with the sequence of snapshots of the heap, an excellent mental grasp of the dheap operations is obtained. Implementing this dheap animation has certainly given us a greater appreciation of the niceties of dheap structure.

The last area of the screen is somewhat more cluttered than the others. In the first frame, we show the "older" table form of the original graph - see the top window for the "new look". Also in the last frame (14), the final result is tabularized. In the other frames, a simple log of major code events is shown. The top two windows show

the state immediately before the first line of the log was executed, and the last line of the code in the frame was done before the windows of the next frame are produced. Again, this was done to be able to represent the dynamics of the algorithm in a static form, on paper. Lines of the form "VERTEX n" or "EDGE m" show the point in the Prim algorithm at which that element is first considered. Lines prefixed with "enter" or "exit" show the calls made by the main Prim algorithm on the dheap control routines. The other lines which occur are those reading "CHANGING EDGE(v,w) TO BLUE" (or RED), indicating the decision made as to the edge's candidacy for membership in the MST.

We will now consider the PASCAL implementation of the Prim algorithm and the dheap routines (see Figure 11), and examine the augmentations. Appendix C specifies the various graphics externals used by the routines. Each requires a "window" parameter: an integer indicating the window (screen area) in which the symbol is to be displayed - if zero, that screen is not displayed. Each also has a "color" parameter, which may be a color word - a predefined constant, e.g. RED:=1, GREEN:=2, etc. to draw or BLACK(:=0) to erase the symbol. Our implementation on the Zenith used the colors to advantage, but the sample animation of Figure 12 was produced on our monochrome APPLE IIe - while the

important features are clear in black and white, the color is much more effective: for example we could not show both the blue and red edges in the graph of the top window.

The code domain animation is accomplished through the use of three routines: SHOWCALL, SHOWRETURN, and SHOWLINE. SHOWCALL and SHOWRETURN count levels of nesting, and display their single string parameter accordingly indented. SHOWCALL prefixes the string with "enter "; SHOWRETURN prefixes the string with "exit ". Each procedure of the program being studied must have SHOWCALL as the first executable statement, and SHOWRETURN as the last, each supplied with an appropriate parameter. One can observe the utility of the code-to-data relation in the SHOWCALLs of INSERT and SHOWRETURNs of DELETEMIN. SHOWLINE is used for display of other interesting areas within the code. Note our use of it in the body of PRIM, to show the points of consideration of the next vertex or edge, as well as the edge-color-changing decision. It should be mentioned that the other lines in the lower window were generated by the outer program, not by use of SHOWLINE.

To achieve display of the dheap, two special routines were written: DHEAPGENXY and DRAWDHEAPNODE. Based on the regular geometry of the "natural" representation of the

d-tree, DHEAPGENXY computes the (x,y) coordinates of the nodes of the dheap, and stores them within the dheap structure itself. DRAWDHEAPNODE draws a specific node of a dheap, showing its heap index and key, and a line connecting it with its parent, if it has one. The augmentation is more complex than the code, but changes are only needed within the several dheap control routines, not within the outer calling routines. We are aware of some loopholes in our implementation, but they are not significant to the current problem being studied. In both DSIFTDOWN and DSIFTUP, calls are made to DRAWDHEAPNODE, drawing and erasing nodes to indicate the shifting of the dheap data downwars or upwards. DINSERT and DDELETEMIN required no alterations, as their animations were achieved by routines they called. DDELETE was a special case - to properly animate the deletion sequence, we resorted to calling the lower level routines used by DRAWDHEAPNODE. We wanted to try showing the shifting in the usual cartoon-like animation, by incremental shifts and redraws, but the available language/system support and graphics could not handle it.

The representation of the "blue tree" in the upper window is achieved through only one routine: DRAWEDGE, called in the PRIM procedure. We have already stored the

(x,y) coordinates of the vertices in the graph structure, and only need to draw the segments which connect them appropriately. Again, red edges are not displayed in the sample, only blue ones, except for the first frame, which shows the entire graph - actually "uncolored". The coordinates are the vertices of a regular n-gon fitting inside the window; these are chosen as they render visible all possible edges of the graph. Arrowheads are used to show the directions of the edges, and the weight of each edge is placed next to the arrowhead. The vertices are also numbered. This documentation of representations, linking them to the data actually used, is very important, and it is one of the more difficult parts to code, because of keeping the diagram uncluttered.

THE FUTURE OF ANIMATION

We feel that animation will definitely become more widespread in the future. As program development environments and high-resolution color graphics terminals become more available, the techniques we have discussed will be of greater importance. With a view towards the future, then, we will give some indications of less obvious applications, and summarize the areas not yet well-defined.

The most significant future use of animation is as a component tool within a program development environment. Its features, enhancing the debugging capabilities available, are conformable with the style found in current PDE's, and we expect that future will see incorporation of animation commonly within PDEs.

Nowithstanding our comments on teaching, debugging, or algorithm analysis, we are extremely fascinated by the application of animation to semantics, especially the "linked forest" semantics described by Culik [4]. Some of our comments have indicated an underlying universal animation scheme for both code and data; this approach should aid in the study of semantics of "live" programs - possibly most exciting for "extensible" languages.

The area of animation in which much work is still needed is that of the problem domain. There are several philosopical implications to be studied, regarding the nature of a "problem-solving form" as a concept, an algorithm as a formal expression of that concept, but still in the abstract plane, and a program as a concrete realization of the algorithm. We perceive this triad of items in the same sense as equivalents in the artistic realm - but as yet no good tools are available to study the artist's emotional thought, as it relates to the artistic prototype (still in the artist's mind), and the final expression in physical media. With algorithm animation, we finally have a tool for exploration of the equivalents in the mathematical realm. We would urge the exploration of the problem domain as a means of approaching the question of the algorithm versus the problem's solution; also the development of a formalism to express common classes of algorithms (sorting, networks, etc.) and subproblems.

The various combinations (code-to-data, etc.) are another area to be described more formally. Various trial animations should be attempted, to determine the most useful features for each application.

CONCLUSIONS

We have attempted to define animation of algorithms, show its uses, give an abstract model and a concrete example, and examine the design criteria in producing a system to perform these animations. We have given a concise collection of facts and suggestions relating to these objectives - while the reader may see many gaps and shortcomings of our discussion, we are aware of many also. A complete treatment on the formal abstract level, the system design level, or implementation level may be forthcoming at some future time. We hope that this work will interest others to attempt animation of their programs, especially those to be used in course instruction. One is reminded of the poem:

"It's a very ancient saying,
But a true and honest thought:
That if you become a teacher,
By your pupils you'll be taught". [15]

As we implemented the dheap animation, we found our understanding of dheaps greatly increased. In fact, our study and animation of the four domains has substantially altered our consciousness of programming in general, and increased our insight into the art and science of computing.

```
PROCEDURE bubblesort(VAR a:ARRAY[1..maxsize] OF INTEGER;
                         n:INTEGER);
{
    Language:    PASCAL
    Status:      UNCOMPILED
    Written by:  P. Floriani  June 6, 1985
    Purpose:     Bubblesort: Original routine to be animated.
                 Sort the first 'n' elements of array 'a'
                 into ascending order, in place
}

 VAR
     i,j,t:INTEGER;
 BEGIN

 FOR i:=1 to n-1 DO
    BEGIN
    FOR j:=i+1 to n DO
       BEGIN
       IF a[i]>a[j] THEN
          BEGIN
          {swap a[i] with a[j]}
          t:=a[i];
          a[i]:=a[j];
          a[j]:=t;
          END{IF};
       END{FOR j};
    END{FOR i};

END{bubblesort};
```

Figure 1 - "bubblesort" original code

```
PROCEDURE bubblesort(VAR a:ARRAY[1..maxsize] OF INTEGER;
                         n:INTEGER);
{
   Language:    PASCAL
   Status:      UNCOMPILED
   Written by:  P. Floriani  June 6, 1985
   Purpose:     Bubblesort: code animation
                Sort the first 'n' elements of array 'a'
                into ascending order, in place
}

VAR
    i,j,t:INTEGER;
BEGIN
                                AA('enter BUBBLESORT(a,n)',2);
                                AA('FOR i:=1 to n-1 DO BEGIN',1);
FOR i:=1 to n-1 DO
   BEGIN
                                AA('FOR j:=i+1 to n DO BEGIN',1);
   FOR j:=i+1 to n DO
      BEGIN
                                AA('IF a[i]>a[j] THEN BEGIN',1);
      IF a[i]>a[j] THEN
         BEGIN
                                AA('swap a[i] and a[j]',2);
         {swap a[i] with a[j]}
                                AA('t:=a[i];',1);
         t:=a[i];
                                AA('a[i]:=a[j];',1);
         a[i]:=a[j];
                                AA('a[j]:=t;',1);
         a[j]:=t;
                                AA('end swap',2);
         END{IF};
                                AA('END{IF a[i]>a[j]};',1);
                                AA('FOR j:=i+1 to n DO BEGIN',1);
      END{FOR j};
                                AA('END{FOR j}',1);
                                AA('FOR i:=1 to n-1 DO BEGIN',1);
   END{FOR i};
                                AA('END{FOR i}',1);
                                AA('exit BUBBLESORT(a,n)',2);
END{bubblesort};
```

Figure 2 - "bubblesort", augmented to animate code domain

```
... CODE FRAGMENT...
{
    Language:    PASCAL, together with
                 a contrived stack-machine assembly code
    Status:      UNCOMPILED (a fragment)
    Written by:  P. Floriani  June 6, 1985
    Purpose:     detailed animation of the code domain
}

{Sample statement:}

                    A:=B*C+D;

{A possible compilation might be:}


    {animation code}                          {object code}

    AA('A:=B*C+D');
    AAI('load B');

                                              load(B);
    AAI('load C');

                                              load(C);
    AAI('multiply');

                                              multiply;
    AAI('term B*C complete')
    AAI('load D');
                                              load(D);
    AAI('add')
                                              add;
    AAI('expression B*C+D complete')
    AAI('store A')
                                              store(A);
    AAI('statement A:=B*C+D complete')
```

Figure 3: Low order code animation

```
PROGRAM FIGURE4;
{
  Language:    PASCAL
  Status:      UNCOMPILED
  Written by: P. Floriani  June 11, 1985
  Purpose:     a sample of data domain animation
}
VAR
    x,y:INTEGER;
BEGIN
                create(x);    {x now exists as a variable}
                create(y);    {y now exists as a variable}
x:=3;
                lvalue(x,3);
                              {specifies that x now has the
                              numeric value three}

y:=x+7;
                lvalue(y,rvalue(x)+7);
                              {specifies that the value of y
                              is now the sum of the current
                              value of x and seven}
END.
```

Figure 4: A simple animation of the data domain

```
PROGRAM FIGURE5;
{
  Language:    PASCAL
  Status:      UNCOMPILED
  Written by: P. Floriani  June 11, 1985
  Purpose:     another example of data domain animation
}
VAR
     i,n,sum:INTEGER;
     a:ARRAY [1..20] OF INTEGER;
BEGIN
                    create(i);
                    create(n);
                    create(sum);
                    FOR μ:=1 to 20 DO
                       create(a[μ]);
                    FOR μ:=1 to 20 DO
                      FOR ν:= 1 to 20 DO
                         IF μ <> ν THEN
                             relate(a[μ],a[ν],SAMEARRAY);
n:=7;
                    lvalue(n,7);
                    lvalue(i,1); { initial FOR assignment }
FOR i:=1 to n do
   BEGIN
      a[i]:=i;
                         lvalue(a[rvalue(i)],rvalue(i));
                         replacerelate(i,a,a[rvalue(i)],INDEXES);

      sum:=sum+i;
                         lvalue(sum,rvalue(sum)+rvalue(i));

                         lvalue(i,rvalue(i)+1); { FOR indexing }
   END;
END.
```

Note that in the above, μ and ν are unbound variables,

not occurring elsewhere in the program.

Figure 5: A more complex data domain animation

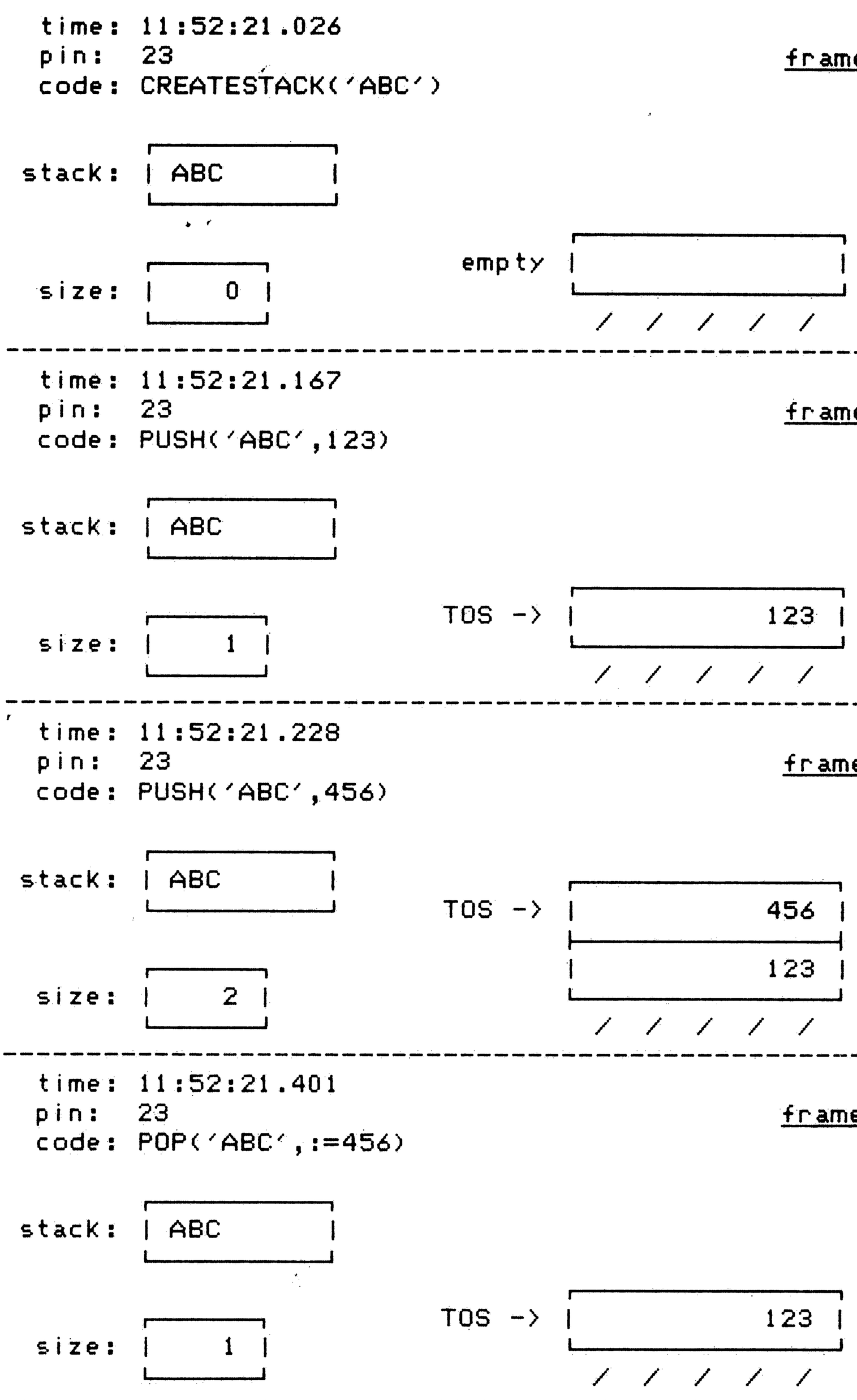


Figure 6: stack control routine animations

```
PROCEDURE CREATESTACK(VAR name:string);
{
    Language:    PASCAL
    Status:      UNCOMPILED
    Written by: P. Floriani  June 12, 1985
    Purpose:     animated front end to CREATESTACK'
                 creates a stack, referenced by 'name'
    Associated: PUSH, POP, KILLSTACK
}
VAR
    s,stacksize,xs,ys:INTEGER;
BEGIN
createstack'(name,q);

stacksize:=0;
newvariable(name & '.size','integer');
setvariable(name & '.size',stacksize);

newscreen(s);
xs:=screen[s].xsize;
ys:=screen[s].ysize;
newvariable(name & '.screen','integer');
setvariable(name & '.screen',s);

standardheading(s,'CREATESTACK(' & name  & ')');

boxstring(s,'stack:',name,8,0,ys/2);
boxinteger(s,'size:',stacksize,4,0,ys/3);

boxstring(s,'empty','                ',
          12,xs/2,stdchar.ysize);
label(s,' /  /  /  /  /',xs/2,0);

END;
```

Figure 7: animation of stack routine CREATESTACK

```
PROCEDURE PUSH(VAR name:string;q:integer);
{
   Language:    PASCAL
   Status:      UNCOMPILED
   Written by: P. Floriani  June 12, 1985
   Purpose:     animated front end to PUSH'
                pushes 'q' onto stack 'name'
   Associated: CREATESTACK, POP, KILLSTACK
}
VAR
   s,stacksize,xs,ys:INTEGER;
BEGIN
push'(name,q);

getvariable(name & '.size',stacksize);
getvariable(name & '.screen',s);
xs:=screen[s].xsize;
ys:=screen[s].ysize;

standardheading(s,'PUSH(' & name & ',' & str(q) & ')');

IF stacksize > 1 THEN
   label(s,'        ',xs/2-stdchar.xsize*7,
         stdchar.ysize*(3*stacksize+1));

boxinteger(s,'TOS ->',q,12,xs/2,
           stdchar.ysize*(3*stacksize+1));

stacksize:=stacksize+1;
setvariable(name & '.size',stacksize);
boxinteger(s,'size:',stacksize,4,0,ys/3);

END;
```

Figure 7 (continued): animation of stack routine PUSH

```
PROCEDURE POP(VAR name:string;VAR q:integer);
{
   Language:    PASCAL
   Status:      UNCOMPILED
   Written by: P. Floriani  June 12, 1985
   Purpose:     animated front end to POP'
                destroys stack 'name'
   Associated: CREATESTACK, PUSH, KILLSTACK
}
VAR
   s,stacksize,xs,ys:INTEGER;
BEGIN
pop'(name,q);

getvariable(name & '.size',stacksize);
getvariable(name & '.screen',s);
xs:=screen[s].xsize;
ys:=screen[s].ysize;

standardheading(s,'POP(' & name & ',' & str(q) & ')');

boxstring(s,'       ','                ',12,xs/2,
          stdchar.ysize*(3*stacksize+1));

stacksize:=stacksize-1;
setvariable(name & '.size',stacksize);
boxinteger(s,'size:',stacksize,4,0,ys/3);

IF stacksize>0 THEN
   label(s,'TOS ->',xs/2-stdchar.xsize*7,
         stdchar.ysize*(3*stacksize+1));

END;
```

Figure 7 (continued): animation of stack routine POP

```
PROCEDURE KILLSTACK(VAR name:string);
{
   Language:    PASCAL
   Status:      UNCOMPILED
   Written by: P. Floriani  June 12, 1985
   Purpose:     animated front end to KILLSTACK'
                destroys stack 'name'
   Associated: CREATESTACK, PUSH, POP
}
VAR
   s,stacksize,xs,ys:INTEGER;
BEGIN
killstack'(name);

getvariable(name & '.size',stacksize);
getvariable(name & '.screen',s);
xs:=screen[s].xsize;
ys:=screen[s].ysize;

standardheading(s,'KILLSTACK(' & name  & ')');

label(s,'stack ' & name ' was deallocated',0,ys/2);

dropscreen(s);
killvariable(name & '.size');
killvariable(name & '.screen');

END;
```

Figure 7 (continued): animation of stack routine KILLSTACK

```
PROGRAM MAIN;
{
    Language:   PASCAL
    Status:     UNCOMPILED
    Written by: P. Floriani  June 13, 1985
    Purpose:    prints distance from a point to an arc
}
FUNCTION distance(x,y,xx,yy:REAL):REAL;
{
    Return Pythagorean distance between (x,y) and (xx,yy)
}
BEGIN
distance:=SQRT((x-xx)*(x-xx)+(y-yy)*(y-yy));
END;

{ function BETWEEN omitted }

    FUNCTION distance2arc(xc,yc,r,sa,ia,xp,yp:REAL):REAL;
    {
      Return the distance between the point (xp,yp) and the
      arc about (xc,yc) of radius 'r' which starts at angle
      'sa' radians, and subtends 'ia' radians
    }
    BEGIN
    IF between(xp-xc,yp-yc,sa,ia) THEN
       BEGIN
       d:=distance(xp,yp,xc,yc);
       IF d < r THEN
          distance2arc:=r-d
       ELSE
          distance2arc:=d-r;
       END
ELSE
       BEGIN
       d1:=distance(xp,yp,xc+r*COS(sa),yc+r*SIN(sa));
       d2:=distance(xp,yp,xc+r*COS(sa+ia),yc+SIN(sa+ia));
       IF d1 < d2 THEN
          distance2arc:=d1
       ELSE
          distance2arc:=d2;
       END;
END;

{ M A I N   P R O G R A M }
BEGIN
WRITELN(distance2arc(2.5,2.5,0.5,180.0,45.0,1.0,3.0));
END;
```

Figure 8: the difficulties of code-to-data animation

```
procedure minspantree(set vertices,vertex s);
(*
   Language:    a variation of SETL
   Status:      UNCOMPILED, copied from uncompiled original
   Written by:  R. E. Tarjan
   Purpose:     solves the minimum spanning tree problem by
                use of Prim's algorithm
   Source:      "Data Structures and Network Algorithms" by
                Robert Endre Tarjan
   Externals:   dheaps: MAKEHEAP, INSERT, SIFTUP, DELETEMIN

The three unbound variables "edges", "cost", and "blue"
should also be notated as parameters to this routine;
the variable is common to the dheap routines; "w" is local.

Input:
   vertices: the set of vertex identifiers
   edges:    the set of edges as vertex pairs (endpoints)
   cost:     function giving weight of an edge
   s:        a distinguished vertex, the "source"

Output:
   blue: for each vertex, either undefined, or a blue edge
         (in the minimum spanning tree) incident to it
*)
vertex v;
heap h;

for v ∈ vertices →
   key(v):=∞
rof;

h:=makeheap({});
v:=s;
do v ≠ null →
   key(v):= -∞;
   for {v,w} ∈ edges(v): cost(v,w) < key(w) →
      key(w):=cost(v,w);
      blue(w):={v,w};
      if not (w ∈ h) →
         insert(w,h);
      | w ∈ h →
         siftup(w,h⁻¹(w),h)
      fi
   rof;
   v:=deletemin(h)
od
end minspantree;
```

Figure 9: Tarjan's code for Prim's algorithm

```
(*
   Language:    a variation of SETL
   Status:      UNCOMPILED, copied from uncompiled original
   Written by: R. E. Tarjan
   Purpose:     dheap manipulation routines
   Source:      "Data Structures and Network Algorithms" by
                Robert Endre Tarjan
*)
integer procedure minchild(integer x,heap h);
return
   if x=|h| → 0
   |  x≠|h| →
      min{d*(x-1)+2..min{d*x+1,|h|}} by keyoh
   fi
end minchild;

procedure siftdown(item i,integer x,
                           modifies heap h);
integer c;
c:=minchild(x,h);
do c≠0 and key(h(c))<key(i) →
   h(x),x,c:=h(c),c,minchild(c,d)
od;
h(x):=i
end siftdown;

procedure siftup(item i,integer x,
                         modifies heap h);
integer p;
p:=⌈(x-1)/d⌉;
do p≠0 and key(h(p))>key(i) →
   h(x),x,p:=h(p),p,⌈(p-1)/d⌉
od;
h(x):=i
end siftup;

procedure delete(item i,modifies heap h);
item j;
j:=h(|h|);
h(|h|):=null;
if i≠j and key(j)≤key(i) →
   siftup(j,h^-1(i),h)
|  i≠j and key(j)>key(i) →
   siftdown(j,h^-1(i),h)
fi
end delete;
```

Figure 10: Tarjan's dheap control routines

```
(*
   Language:    a variation of SETL
   Status:      UNCOMPILED, copied from uncompiled original
   Written by: R. E. Tarjan
   Purpose:     dheap manipulation routines
   Source:      "Data Structures and Network Algorithms" by
                Robert Endre Tarjan
*)
item function deletemin(modifies heap h);
if h={} →
   return null
|  h≠{} →
   item i;
   i:=h(1);
   delete(h(1),h);
   return i;
fi
end deletemin;

procedure insert(item i,modifies heap h);
siftup(i,|h|+1,h)
end insert;

heap function makeheap(set s);
map h;
h:={};
for i ∈ s →
   h(|h|+1):=i
rof;
for x=|s|,|s|-1..1 →
   siftdown(h(x),x,h)
rof
return h
end makeheap;
```

Figure 10 (continued): Tarjan's dheap control routines

```
CONST
   MAXDHEAP={maximum size of a dheap};
   MAXVERT={maximum number of vertices in a graph};
   MAXEDGE={maximum number of edges in a graph};
   INFINITY={a real number larger than any of the problem};
  {BLACK,RED,BLUE,YELLOW, are also defined here}

TYPE
   DHEAP=RECORD
      D:          INTEGER;  {'d' for this dheap}
      SIZE:       INTEGER;  {number of members}
      WINDOW:     INTEGER;  {window for its display}
      H:          ARRAY [1..MAXDHEAP] OF INTEGER; {pointer}
      HI:         ARRAY [1..MAXDHEAP] OF INTEGER; {inverse}
      KEY:        ARRAY [1..MAXDHEAP] OF REAL;    {dheap key}
      X:          ARRAY [1..MAXDHEAP] OF INTEGER; {x coord}
      Y:          ARRAY [1..MAXDHEAP] OF INTEGER; {y coord}
END;

CLISTPTR=^CLISTNODE; {defines a circular list}

CLISTNODE=RECORD
   THIS:       INTEGER; {integer on the list}
   NEXT:       CLISTPTR; {points to next entry}
END;

CLIST=RECORD
   TAIL:       CLISTPTR; { TAIL^.NEXT is the 'head' }
   COUNT:      INTEGER;  { number of entries on list}
END;

GRAPH=RECORD
   EDGES:      INTEGER;  {number of edges}
   VFROM:      ARRAY [1..MAXEDGE] OF INTEGER;{from vertex}
   VTO:        ARRAY [1..MAXEDGE] OF INTEGER;{to vertex}
   COST:       ARRAY [1..MAXEDGE] OF INTEGER;{edge weight}
   VERTICES:  INTEGER;  {number of vertices}
   VERTEX:     ARRAY [1..MAXVERT] OF INTEGER; {ident}
   EIN:        ARRAY [1..MAXVERT] OF CLIST; {edges in}
   EOUT:       ARRAY [1..MAXVERT] OF CLIST; {edges out}
   EINC:       ARRAY [1..MAXVERT] OF CLIST; {incident}
   X:          ARRAY [1..MAXVERT] OF INTEGER; {x coord}
   Y:          ARRAY [1..MAXVERT] OF INTEGER; {y coord}
END;

VAR                    {the windows}
   WXL,WYL,WXH,WYH: ARRAY [1..MAXWINDOWS] OF INTEGER;
```

Figure 11: Prim (animated) Global declarations

```
PROCEDURE PRIM(VAR G:GRAPH;WGRAPH,WDHEAP:INTEGER);
{
   Language:    PASCAL
   Status:      HAND-COPIED from RUNNING program
   Written by:  P. Floriani  April, 1985
   Purpose:     ANIMATION OF PRIM MINIMUM SPANNING TREE
   Externals:   SHOWLINE,DMAKEHEAP,DSIFTUP,DRAWEDGE,DINSERT,
                DDELETEMIN,N2S
   Globals:     RED,BLUE,INFINITY,DHEAPSIZE

   Parameters: G       the graph
               WGRAPH  the window for display of the graph
               WDHEAP  the window for display of the dheap
}

VAR
   H:DHEAP;
   VBLUE: ARRAY [1..MAXVERT] OF INTEGER;
   I,V,W,VW:INTEGER;
   KOST:REAL;
   EPTR:CLISTPTR;
```

Figure 11 (continued): Prim (animated) main algorithm

```
BEGIN
FOR V:=1 TO G.VERTICES DO
    BEGIN
    H.KEY[V]:=INFINITY;
    VBLUE[V]:=0;
    END;
DMAKEHEAP(H,DHEAPSIZE('d' of the dheap),WHEAP);
V:=1; { we are using vertex 1 as the source }
WHILE V<>0 DO {do v ≠ null}
    BEGIN
    SHOWLINE('VERTEX '+N2S(V));
    H.KEY[V]:=-INFINITY;                {key(v):=-∞}
    EPTR:=G.EINC[V].HEAD^.NEXT;         {for (v,w)∈edges(v)}
    FOR I:=1 TO G.EINC[V].COUNT DO
        BEGIN
        VW:=EPTR^.THIS;
        SHOWLINE('EDGE ('+N2S(G.VFROM[VW])+','
                         +N2S(G.VTO[VW])+')');
        IF V=G.VFROM[VW] THEN
            W:=G.VTO[VW]
        ELSE
            W:=G.VFROM[VW];
        KOST:=H.KEY[W];
        IF G.COST[VW]<KOST THEN         {:cost(v,w)<key(w)→}
            BEGIN
            H.KEY[W]:=G.COST[VW];       {key(w):=cost(v,w)}
            IF VBLUE(W)<>0 THEN         {blue(w):=(v,w)}
                BEGIN
                DRAWEDGE(G,VBLUE[W],RED,WGRAPH);
                SHOWLINE('CHANGING EDGE ('+
                        N2S(G.VFROM(VBLUE[W]))+','+
                        N2S(G.VTO(VBLUE[W]))+') TO RED');
                END;
            VBLUE[W]:=VW;
            DRAWEDGE(G,VBLUE[W],BLUE,WGRAPH);
            SHOWLINE('CHANGING EDGE ('+N2S(G.VFROM(VBLUE[W]))
                    +','+N2S(G.VTO(VBLUE[W]))+') TO BLUE');
                        {if not (w∈h) → insert(w,h)}
            IF ABS(KOST)=INFINITY THEN DINSERT(W,H)
                        {| w∈h → siftup(w,h⁻¹(w),h)}
            ELSE DSIFTUP(W,H.HI[W],H);
        END{IF G.COST[VW]};
    EPTR:=EPTR^.NEXT;
    END{FOR};               {rof}

    V:=DDELETEMIN(H);
    END{WHILE V<>0};        {od}
END;
```

Figure 11 (continued): Prim (animated) main algorithm

```
PROCEDURE DHEAPGENXY(VAR H:DHEAP);
{
    Language:    PASCAL
    Status:      HAND-COPIED from RUNNING program
    Written by:  P. Floriani  April, 1985
    Purpose:     this routine computes the x,y locations of
                 the nodes of the dheap, storing them in the
                 heap.
    Externals:   NUMW
    Globals:     WXL,WYL,WXH,WYH,YELLOW
}
VAR
    W,LEVEL,POS,I,PWRD:INTEGER;
    MAXLEVEL,DX,DY,X,Y,LX,LY:INTEGER;
BEGIN
W:=H.WINDOW;
LX:=WXH[W]-WXL[W];
LY:=WYH[W]-WYL[W];

NUMW(LX-16,LY-12,H.D,2,YELLOW,W); {display 'd'}

DX:=LX DIV 2;
MAXLEVEL:=LY DIV 30;
DY:=LY DIV MAXLEVEL;

LEVEL:=0;
POS:=0;
PWRD:=1;

FOR I:=1 TO H.SIZE DO
    BEGIN
    {do node I of dheap}
    H.X[I]:=DX*(POS+1);
    H.Y[I]:=DY*(MAXLEVEL-LEVEL);
    POS:=POS+1;
    IF POS=PWRD THEN
        BEGIN
        POS:=0;
        LEVEL:=LEVEL+1;
        PWRD:=PWRD*H.D;
        DX:=LX DIV (PWRD+1);
        END;
    END;
END;
```

Figure 11 (continued): Prim (animated) DHEAPGENXY

```
PROCEDURE DRAWDHEAPNODE(I:INTEGER;VAR H:DHEAP;C:INTEGER);
{
   Language:    PASCAL
   Status:      HAND-COPIED from RUNNING program
   Written by:  P. Floriani  April, 1985
   Purpose:     this routine draws node I of dheap H in
                color C in the window H.WINDOW
   Externals:   NUMW,PARENT,LINEW,CHARW
   Globals:     INFINITY,INFINITYCHAR,MINUSCHAR,PLUSCHAR,
                BLACK,YELLOW
}
VAR
   XX,YY,P,W,K:INTEGER;
   THISKEY:REAL;
BEGIN
W:=H.WINDOW;
IF W<>0 THEN
   BEGIN
   NUMW(2,WYH[W]-WYL[W]-12,H.SIZE,2,YELLOW,W);

   P:=PARENT(I);
   XX:=H.X[I];
   YY:=H.Y[I];
   IF P<>0 THEN
      LINEW(XX,YY,H.X[P],H.Y[P]-22,C,W);
   NUMW(XX-7,YY-10,H.H[I],2,C,W);
   THISKEY:=H.KEY[H.H[I]];
   IF THISKEY=INFINITY THEN
      BEGIN
      CHARW(XX-7,YY-20,PLUSCHAR,C,BLACK,W);
      CHARW(XX,YY-20,INFINITYCHAR,C,BLACK,W);
      END
   ELSE
      IF THISKEY=-INFINITY THEN
         BEGIN
         CHARW(XX-7,YY-20,MINUSCHAR,C,BLACK,W);
         CHARW(XX,YY-20,INFINITYCHAR,C,BLACK,W);
         END
      ELSE
         BEGIN
         K:=ROUND(THISKEY);
         NUMW(XX-7,YY-20,K,2,C,W);
         END;
   END;
END;
```

Figure 11 (continued): Prim (animated) DRAWDHEAPNODE

```
PROCEDURE DSIFTDOWN(I,X:INTEGER;VAR H:DHEAP);
{
   Language:    PASCAL
   Status:      HAND-COPIED from RUNNING program
   Written by:  P. Floriani  April, 1985
   Purpose:     this routine "sifts" the key of I down in
                the dheap - the motion is animated.
   Externals:   N2S,DMINCHILD,DRAWDHEAPNODE,
                SHOWCALL,SHOWRETURN
   Globals:     BLACK,YELLOW
}
VAR
   C,T1,T2,T3:INTEGER;
   DONE:BOOLEAN;
BEGIN
SHOWCALL('DSIFTDOWN('+N2S(I)+','+N2S(X)+')');

C:=DMINCHILD(X,H);
IF C=0 THEN
   DONE:=TRUE
ELSE
   DONE:=(H.KEY[H.H[C]]>=H,KEY[I]);

IF NOT DONE THEN
   BEGIN
   WHILE NOT DONE DO
      BEGIN
      T1:=H.H[C];
      T2:=C;
      T3:=DMINCHILD(C,H);
      DRAWDHEAPNODE(X,H,BLACK);
      H.H[X]:=T1;H.HI[T1]:=X;
      DRAWDHEAPNODE(X,H,YELLOW);
      X:=T2;
      C:=T3;
      IF C=0 THEN
         DONE:=TRUE
      ELSE
         DONE:=(H.KEY[H.H[C]]>=H.KEY[I]);
      END{WHILE NOT DONE};

DRAWDHEAPNODE(X,H,BLACK);
END{IF NOT DONE};

H.H[X]:=I;H.HI[I]:=X;
DRAWDHEAPNODE(X,H,YELLOW);
SHOWRETURN('DSIFTDOWN');
END;
```

Figure 11 (continued): Prim (animated) DSIFTDOWN

```
PROCEDURE DSIFTUP(I,X:INTEGER;VAR H:DHEAP);
{
    Language:    PASCAL
    Status:      HAND-COPIED from RUNNING program
    Written by:  P. Floriani  April, 1985
    Purpose:     this routine "sifts" the key of I up in
                 the dheap - the motion is animated.
    Externals:   N2S,PARENT,DRAWDHEAPNODE,
                 SHOWCALL,SHOWRETURN
    Globals:     BLACK,YELLOW
}
VAR
    P,T1,T2,T3:INTEGER;
    DONE:BOOLEAN;
BEGIN
SHOWCALL('DSIFTUP('+N2S(I)+','+N2S(X)+')');

P:=PARENT(X,H);
IF P=0 THEN
    DONE:=TRUE
ELSE
    DONE:=(H.KEY[H.H[P]]<=H,KEY[I]);

IF NOT DONE THEN
    BEGIN
    WHILE NOT DONE DO
        BEGIN
        T1:=H.H[P];
        T2:=P;
        T3:=PARENT(P,H);
        DRAWDHEAPNODE(X,H,BLACK);
        H.H[X]:=T1;H.HI[T1]:=X;
        DRAWDHEAPNODE(X,H,YELLOW);
        X:=T2;
        P:=T3;
        IF P=0 THEN
            DONE:=TRUE
        ELSE
            DONE:=(H.KEY[H.H[P]]<=H.KEY[I]);
        END{WHILE NOT DONE};

    DRAWDHEAPNODE(X,H,BLACK);
    END{IF NOT DONE};

H.H[X]:=I;H.HI[I]:=X;
DRAWDHEAPNODE(X,H,YELLOW);
SHOWRETURN('DSIFTUP');
END;
```

Figure 11 (continued): Prim (animated) DSIFTUP

```
PROCEDURE DDELETE(I:INTEGER;VAR H:DHEAP);
{
   Language:    PASCAL
   Status:      HAND-COPIED from RUNNING program
   Written by:  P. Floriani  April, 1985
   Purpose:     this routine deletes the node of I from
                the dheap - the motion is animated.
   Externals:   N2S,PARENT,DRAWDHEAPNODE,LINEW,NUMW,
                SHOWCALL,SHOWRETURN,DSIFTUP,DSIFTDOWN
   Globals:     BLACK,YELLOW
}
VAR
   P,J,W,LX,LY:INTEGER;
BEGIN
SHOWCALL('DDELETE('+N2S(I)+')');

J:=H.H[H.SIZE];

W:=H.WINDOW;
IF W<>0 THEN
   BEGIN
   P:=PARENT(H.SIZE,H);
   IF P<>0 THEN
      LINEW(H.X[H.SIZE],H.Y[H.SIZE],H.X[P],H.Y[P]-22,
            BLACK,W);
   END;

H.SIZE:=H.SIZE-1;

IF I<>J THEN
   IF H.KEY[J]<=H.KEY[I] THEN
      DSIFTUP(J,H.HI[I],H)
   ELSE
      DSIFTDOWN(J,H.HI[I],H);

H.SIZE:=H.SIZE+1;
DRAWDHEAPNODE(H.SIZE,H,BLACK);
H.SIZE:=H.SIZE-1;

IF W<>0 THEN
   BEGIN
   LX:=WXH[W]-WXL[W];
   LY:=WYH[W]-WYL[W];
   NUMW(2,LY-12,H.SIZE,2,YELLOW,W);
   END;

END;
```

Figure 11 (continued): Prim (animated) DDELETE

```
FUNCTION DDELETEMIN(VAR H:DHEAP):INTEGER;
{
   Language:    PASCAL
   Status:      HAND-COPIED from RUNNING program
   Written by:  P. Floriani  April, 1985
   Purpose:     this routine deletes the node with the
                smallest key from the dheap - the motion is
                animated.
   Externals:   SHOWCALL,SHOWRETURN,DDELETE,N2S
}
VAR
   I:INTEGER;

BEGIN
SHOWCALL('DDELETEMIN()');

IF H.SIZE=0 THEN
   I:=0
ELSE
   BEGIN
   I:=H.H[1];
   DDELETE(I,H);
   END;

DDELETEMIN:=I;

SHOWRETURN('DDELETEMIN():='+N2S(I));
END;
```

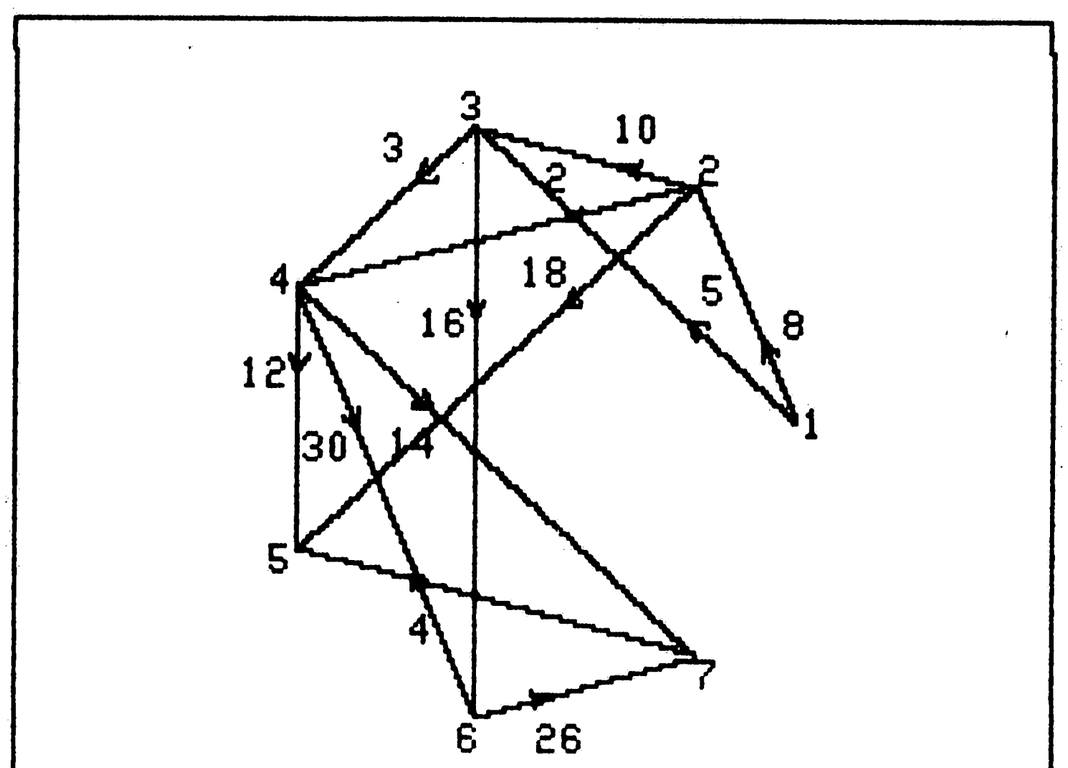
Figure 11 (continued): Prim (animated) DDELETEMIN

```
PROCEDURE DINSERT(I:INTEGER;VAR H:DHEAP);
{
   Language:    PASCAL
   Status:      HAND-COPIED from RUNNING program
   Written by:  P. Floriani  April, 1985
   Purpose:     this routine inserts a new node i' into
                the dheap - the motion is animated.
   Externals:   SHOWCALL,SHOWRETURN,DSIFTUP,N2S,DHEAPGENXY
}
BEGIN
SHOWCALL('DINSERT('+N2S(I)+')');

H.SIZE:=H.SIZE+1;
H.H[H.SIZE]:=I;H.HI[I]:=H.SIZE;
DHEAPGENXY(H);
DSIFTUP(I,H.SIZE,H);

SHOWRETURN('DINSERT');
END;
```

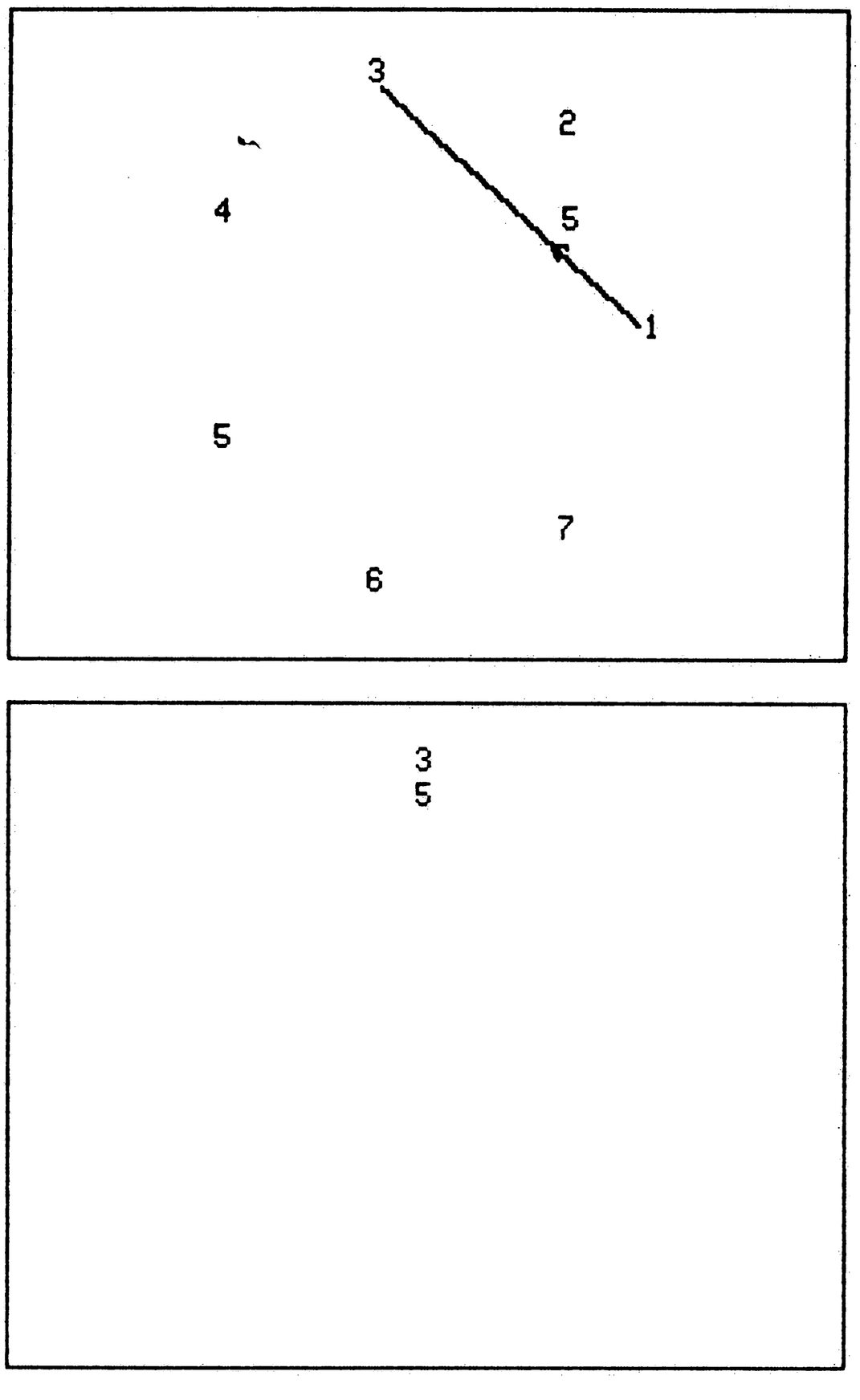
Figure 11 (continued): Prim (animated) DINSERT



EDGE	COST	START	END
1	8	1	2
2	5	1	3
3	10	2	3
4	2	2	4
5	18	2	5
6	3	3	4
7	16	3	6
8	12	4	5
9	30	4	6
10	14	4	7
11	4	5	7
12	26	6	7

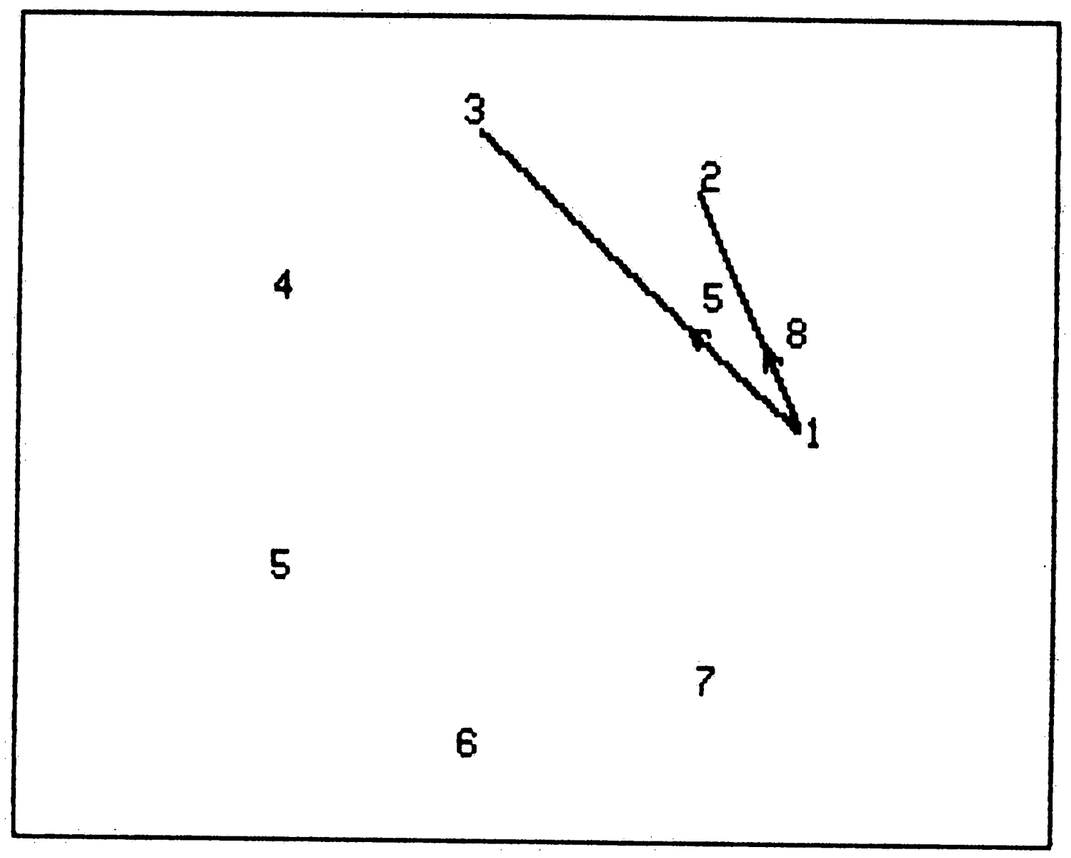
VERTEX 1
EDGE (1,3)
CHANGING EDGE(1,3) TO BLUE
enter INSERT(3)
exit INSERT

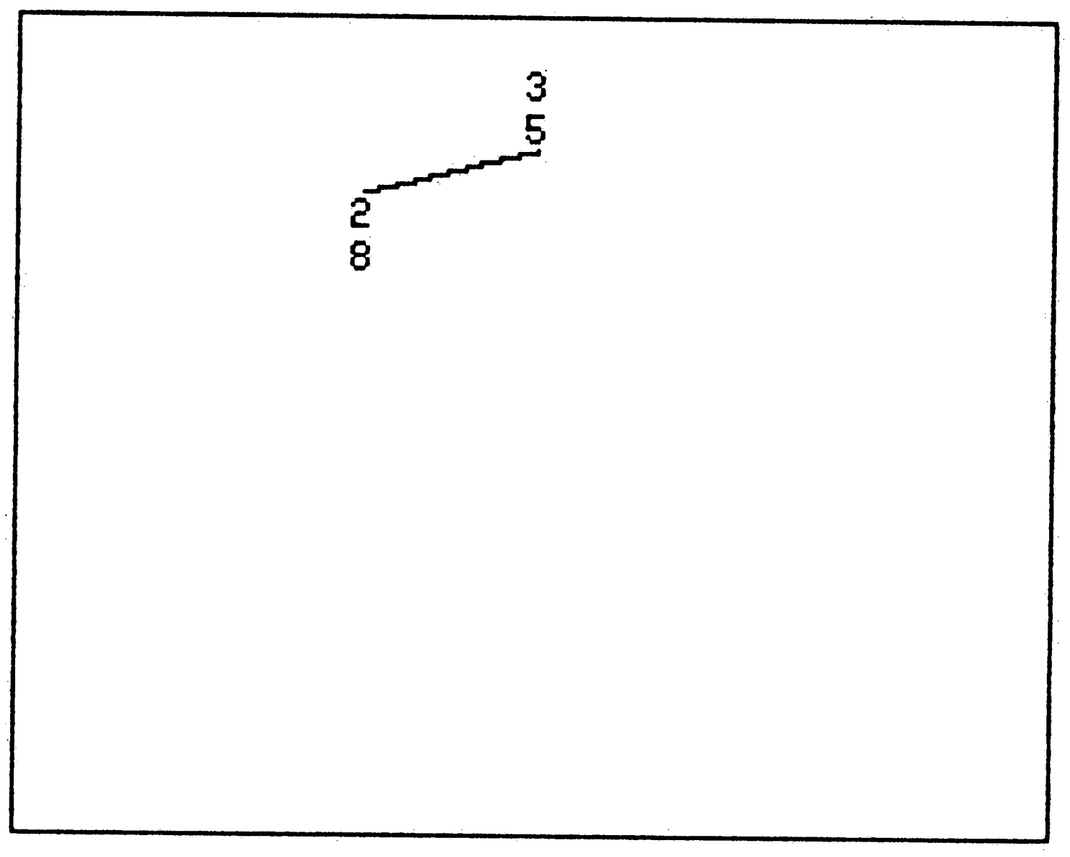
Figure 12: Sample Animation Frame 1




```
EDGE (1,2)
 CHANGING EDGE(1,2) TO BLUE
    enter  INSERT(2)
    exit   INSERT
```

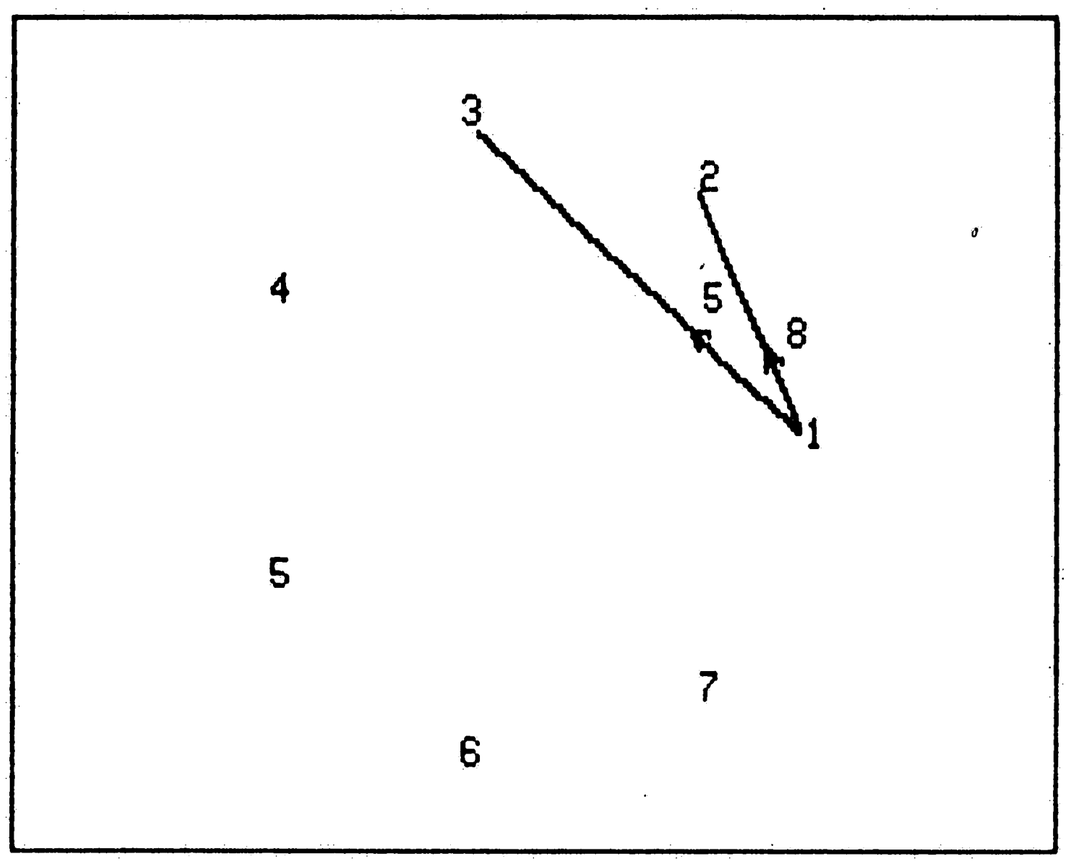
Figure 12: Sample Animation Frame 2

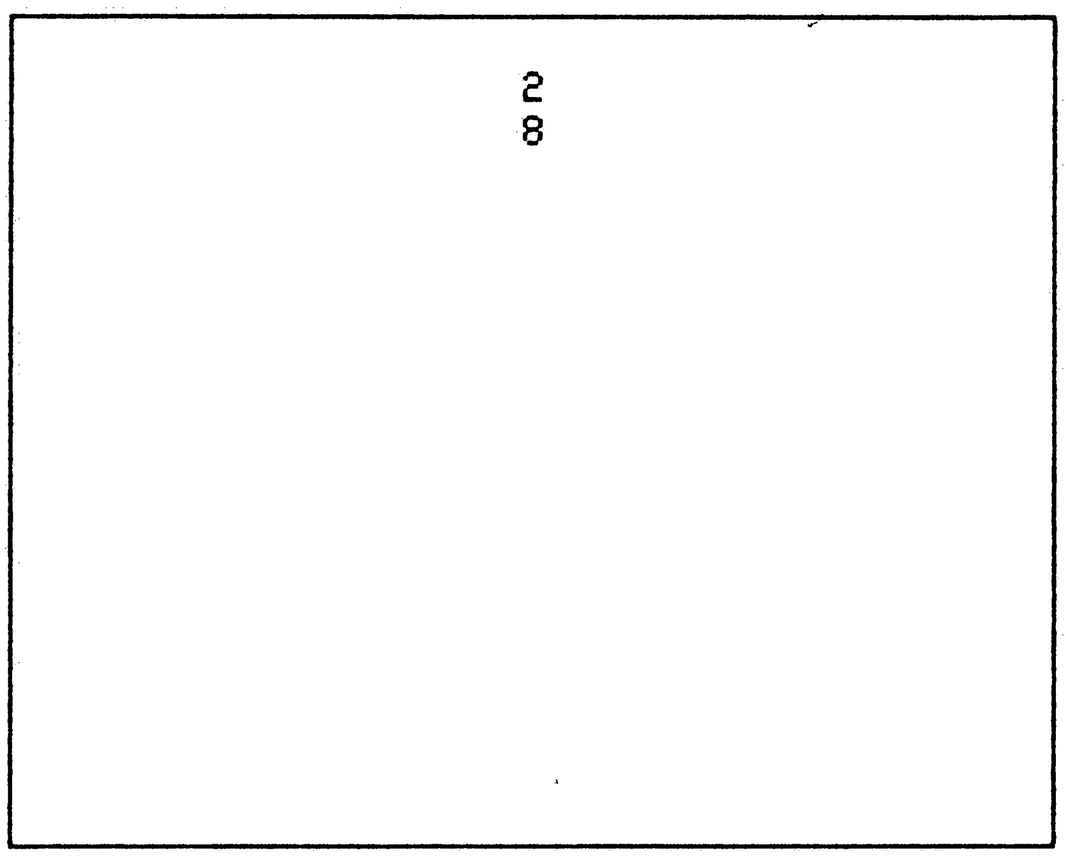





```
enter DELETEMIN
exit  DELETEMIN:=3
```

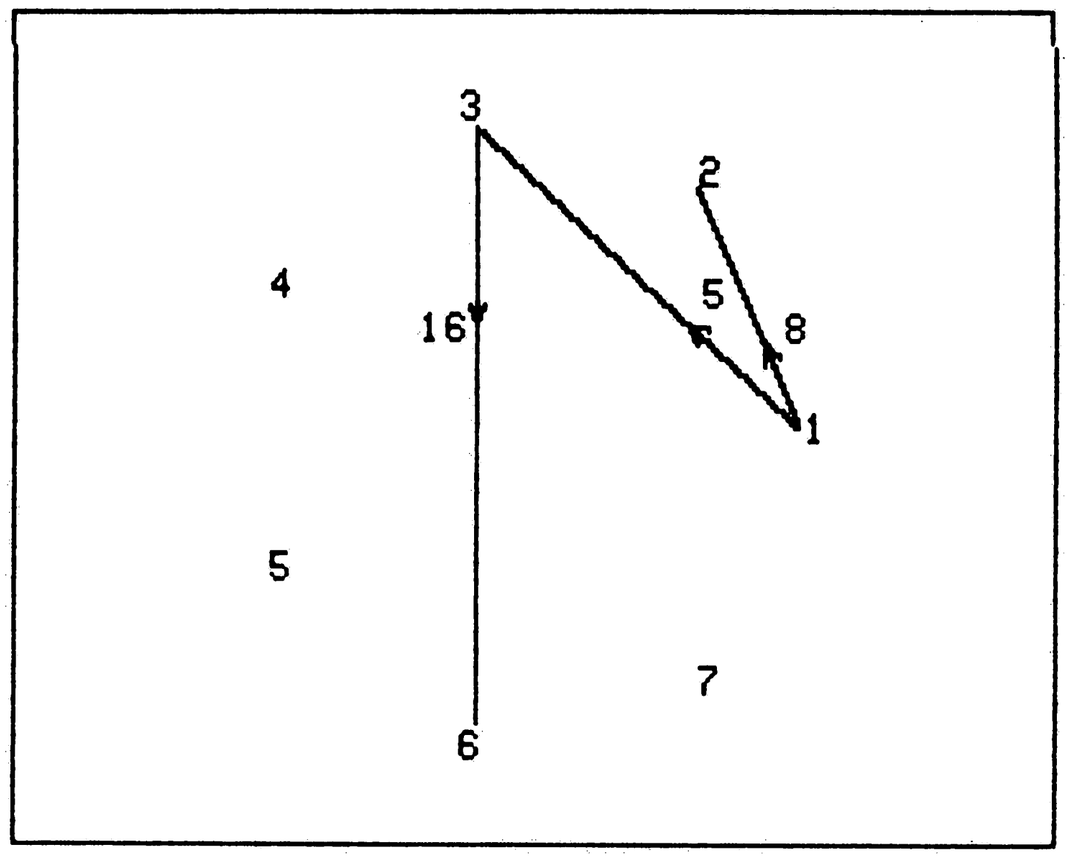
Figure 12: Sample Animation Frame 3

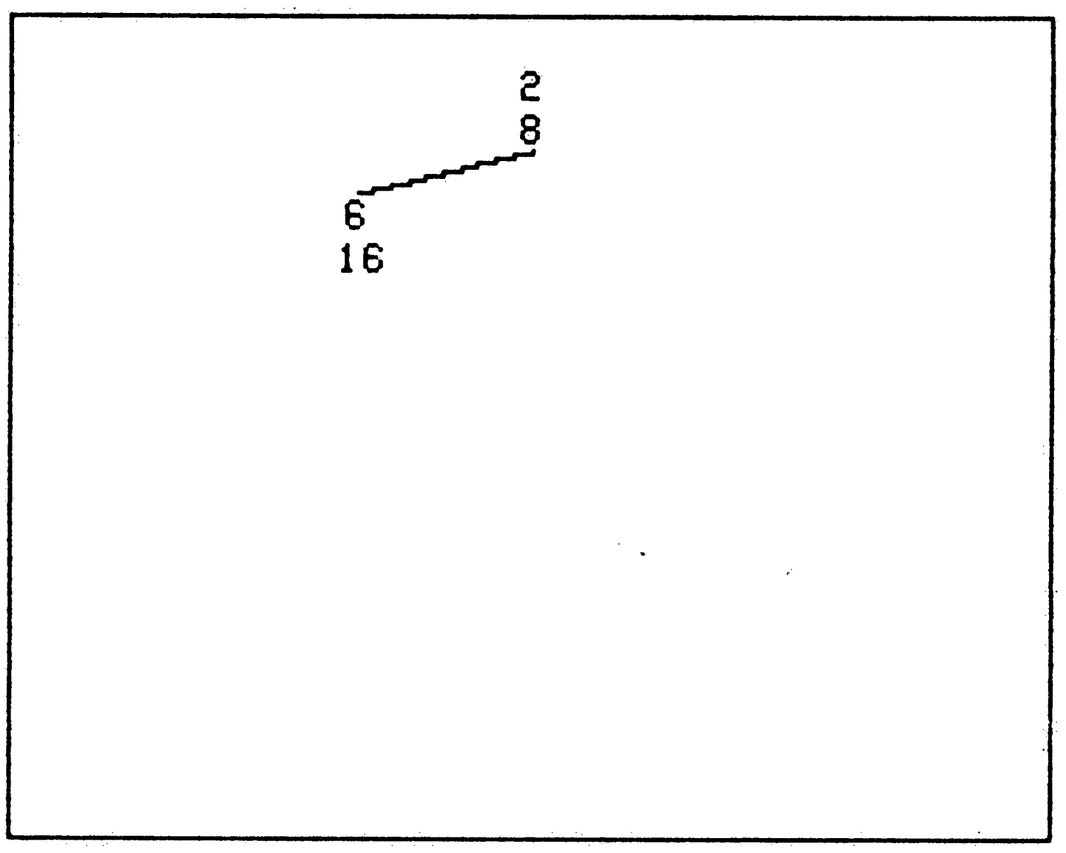





```
VERTEX 3
EDGE (3,6)
 CHANGING EDGE(3,6) TO BLUE
   enter INSERT(6)
   exit  INSERT
```

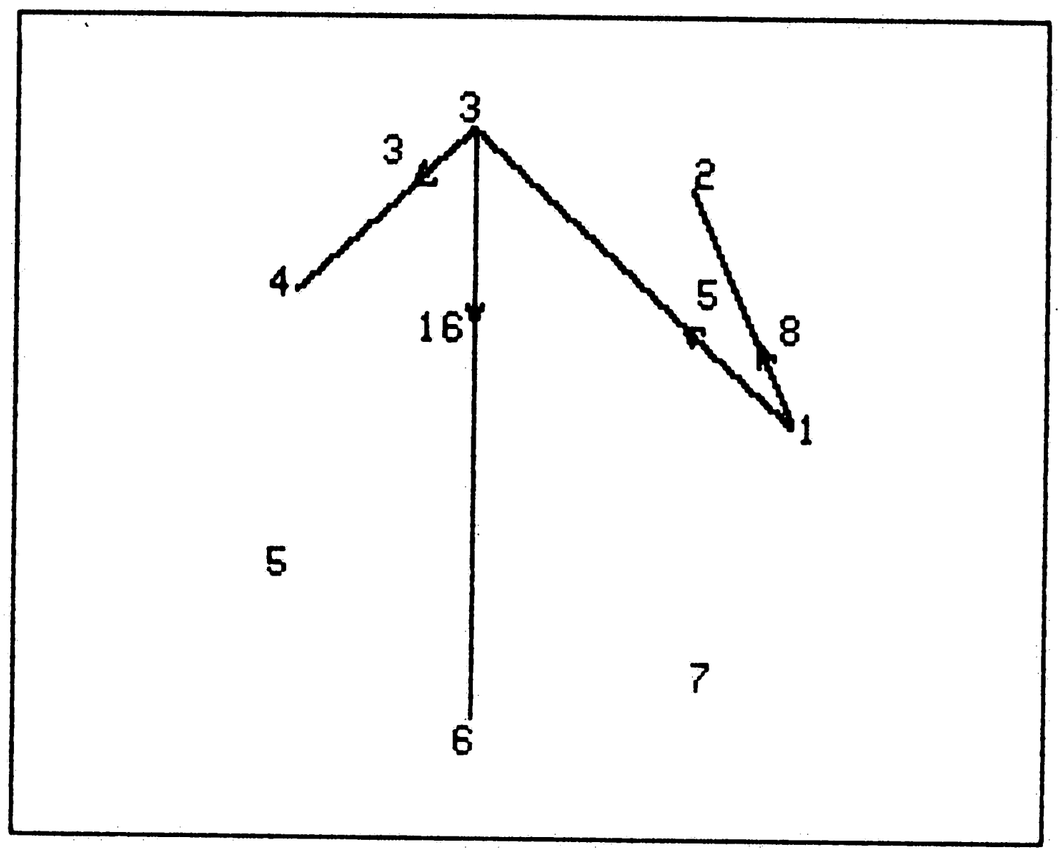
Figure 12: Sample Animation Frame 4

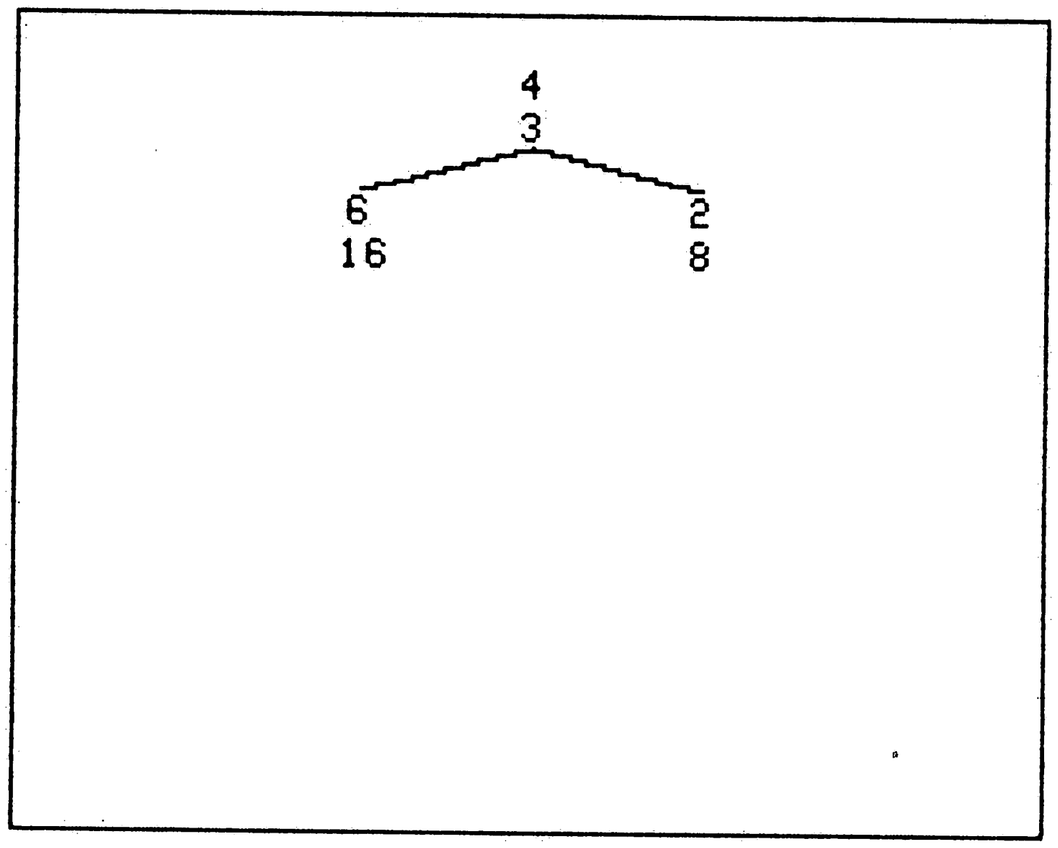





```
EDGE (3,4)
 CHANGING EDGE(3,4) TO BLUE
    enter INSERT(4)
    exit  INSERT
```

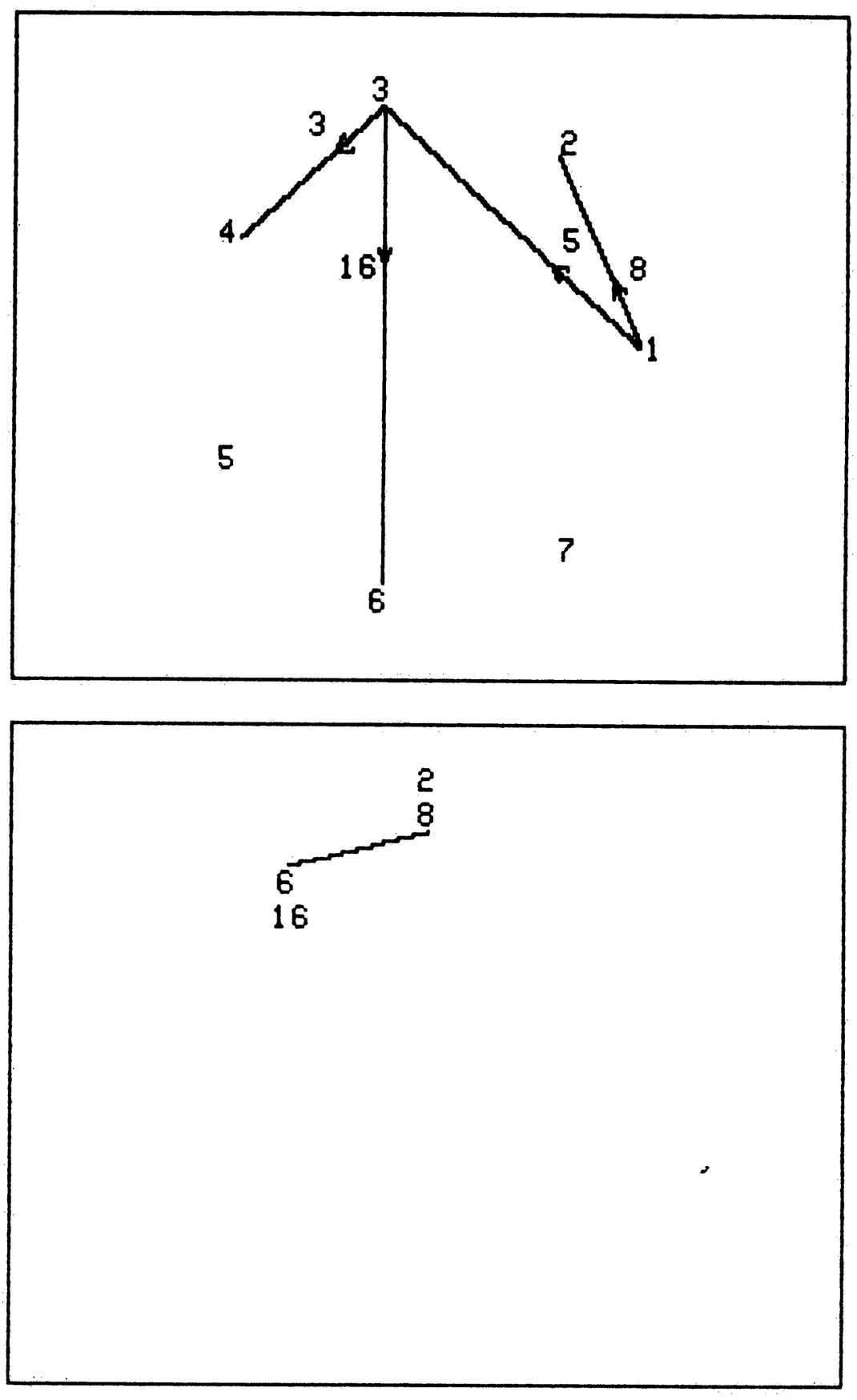
Figure 12: Sample Animation Frame 5





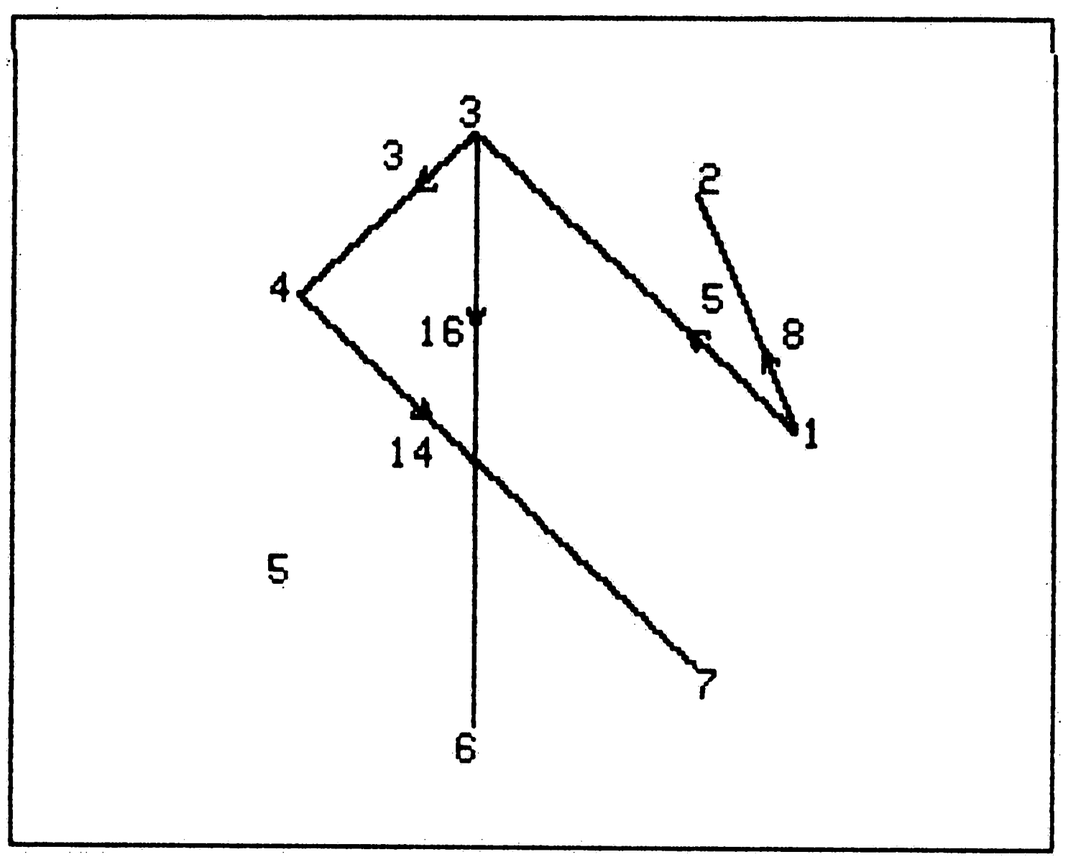

```
EDGE (2,3)
EDGE (1,3)
    enter DELETEMIN
    exit  DELETEMIN:=4
```

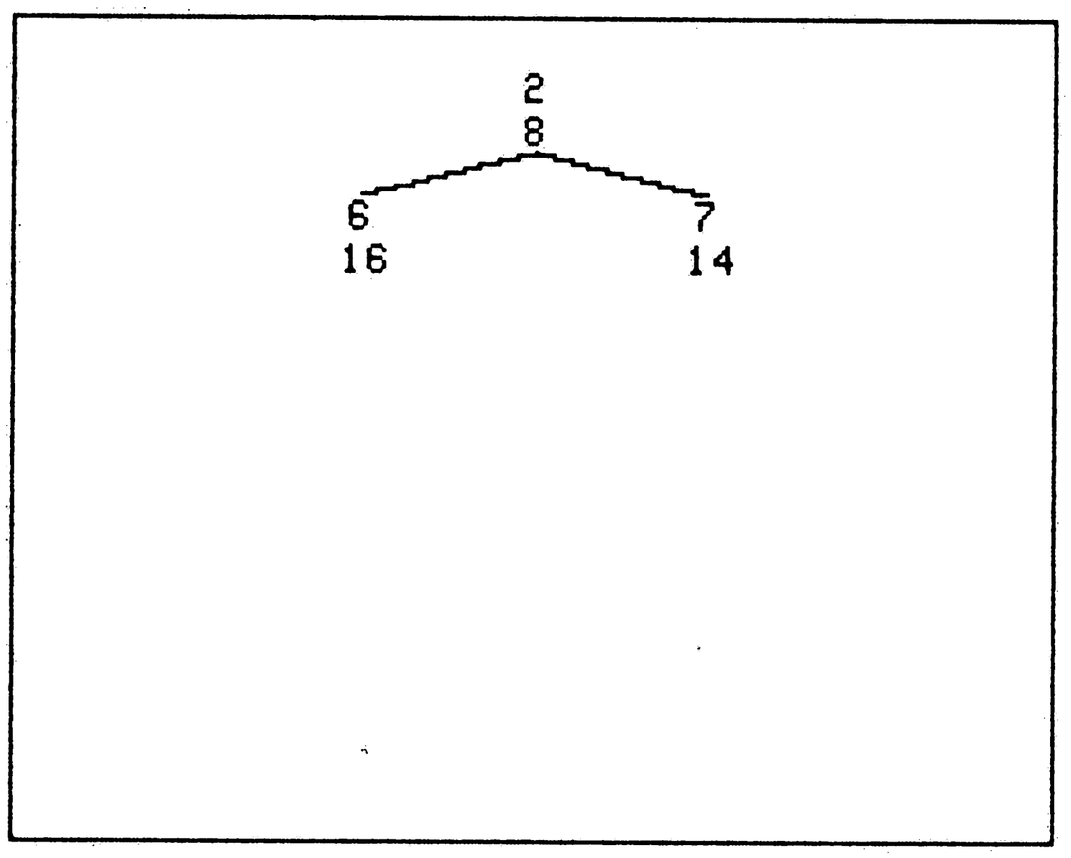
Figure 12: Sample Animation Frame 6




```
VERTEX 4
EDGE (4,7)
 CHANGING EDGE(4,7) TO BLUE
   enter INSERT(7)
   exit  INSERT
```

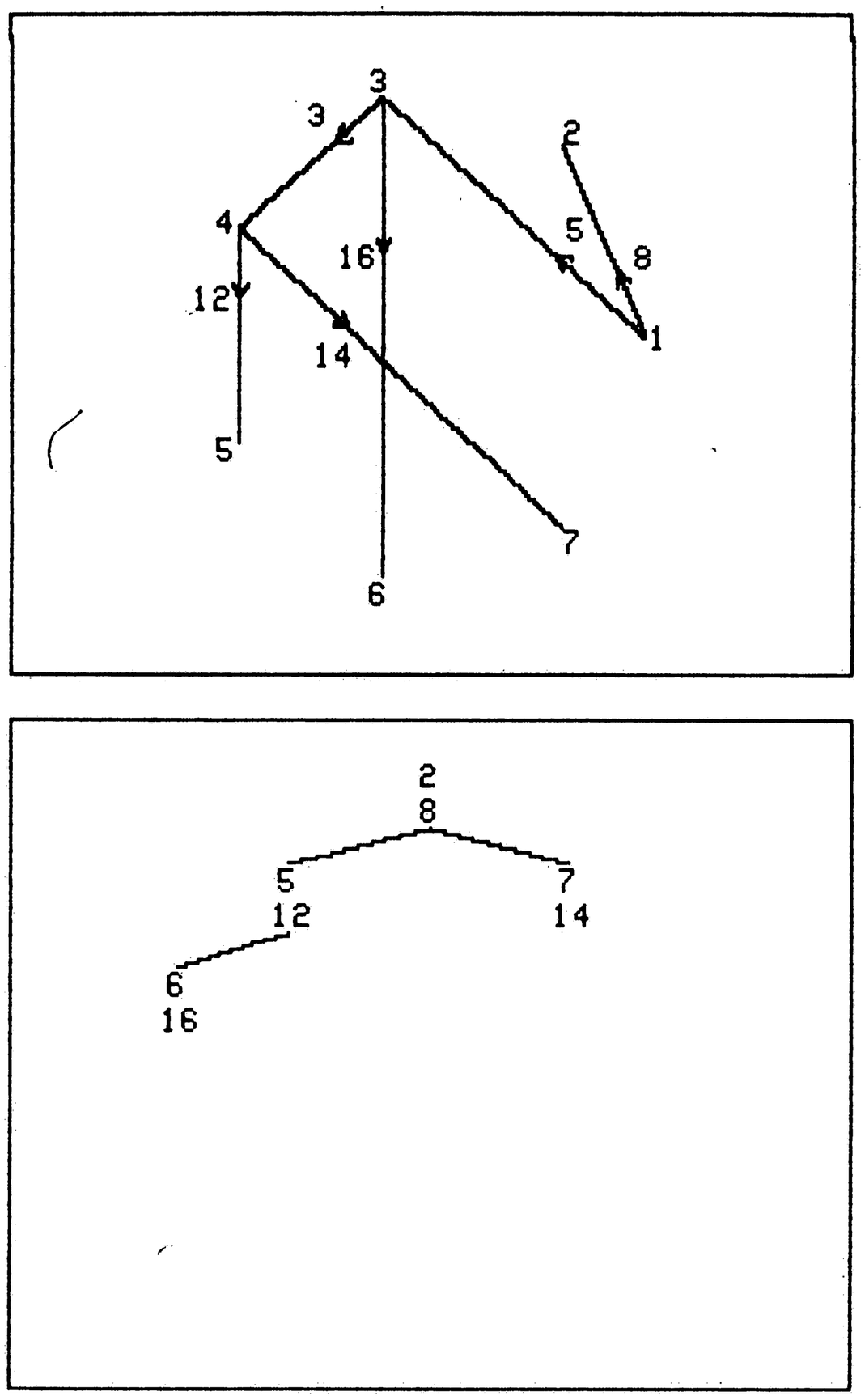
Figure 12: Sample Animation Frame 7





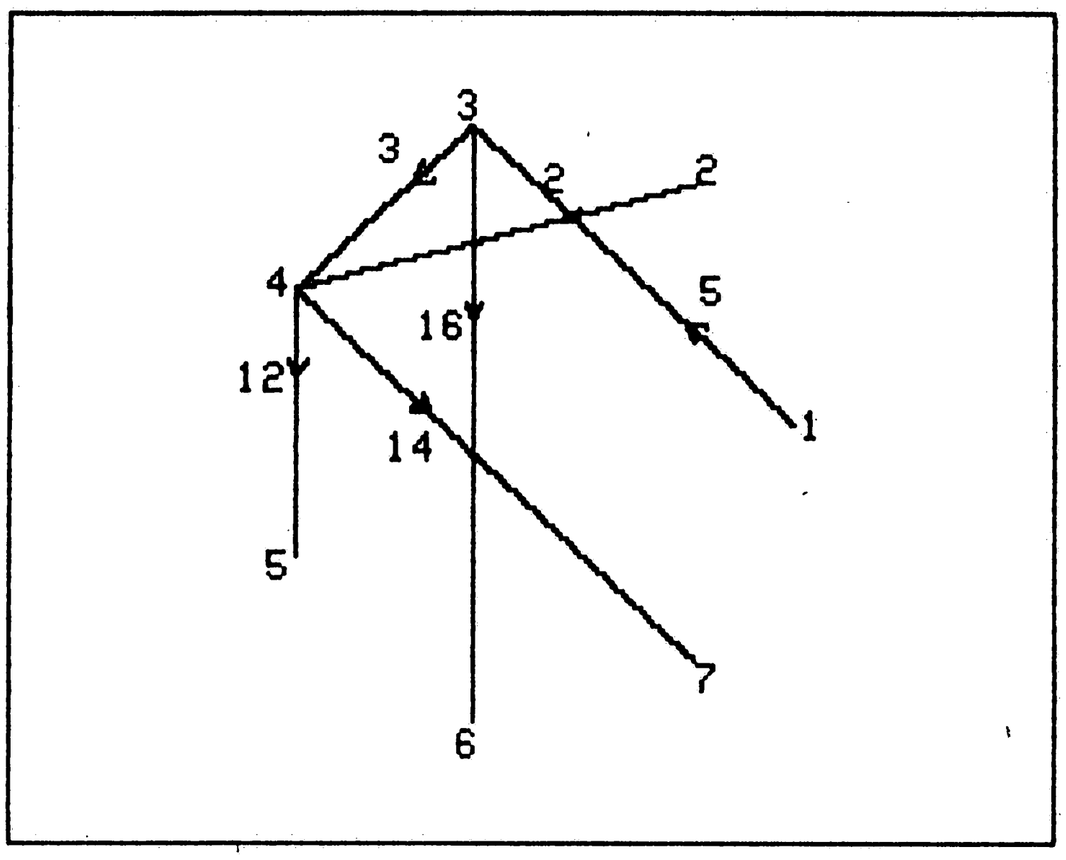

```
EDGE (4,6)
EDGE (4,5)
 CHANGING EDGE(4,5) TO BLUE
    enter INSERT(5)
    exit  INSERT
```

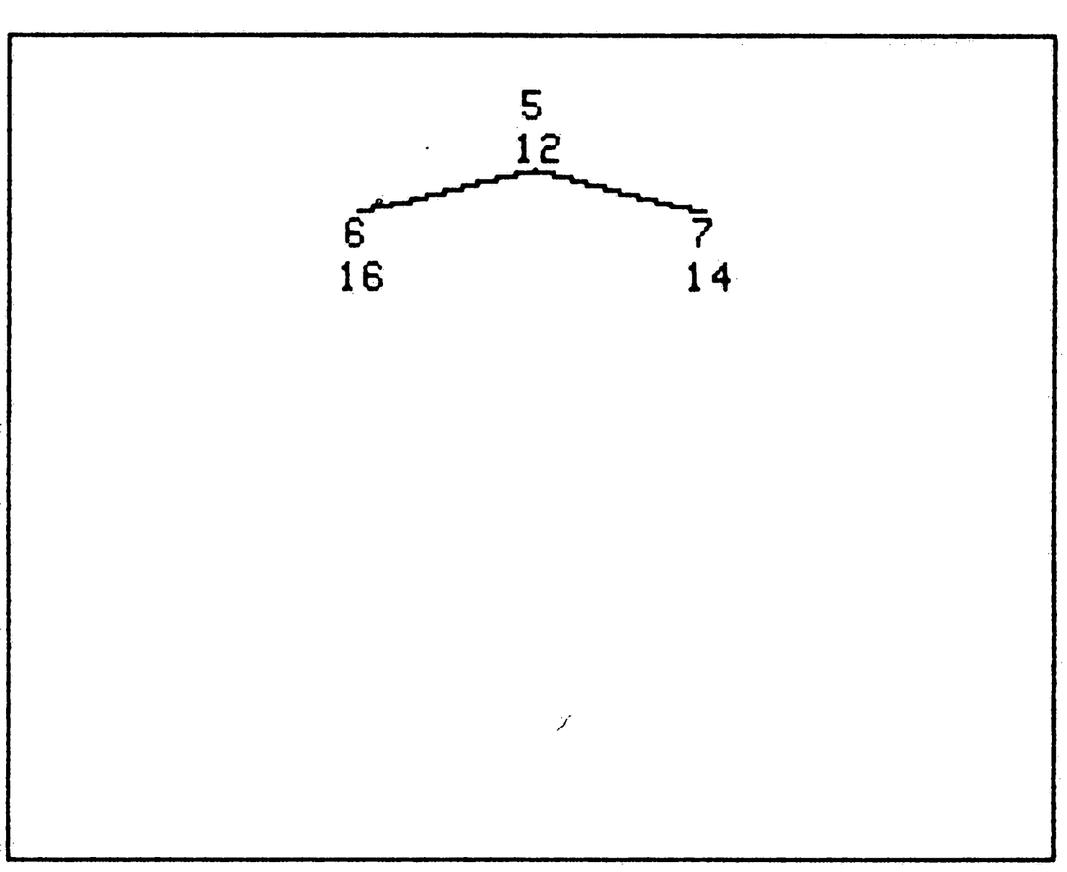
Figure 12: Sample Animation Frame 8




```
EDGE (3,4)
EDGE (2,4)
 CHANGING EDGE (1,2) TO RED
 CHANGING EDGE(2,4) TO BLUE
   enter DELETEMIN
   exit  DELETEMIN:=2
```

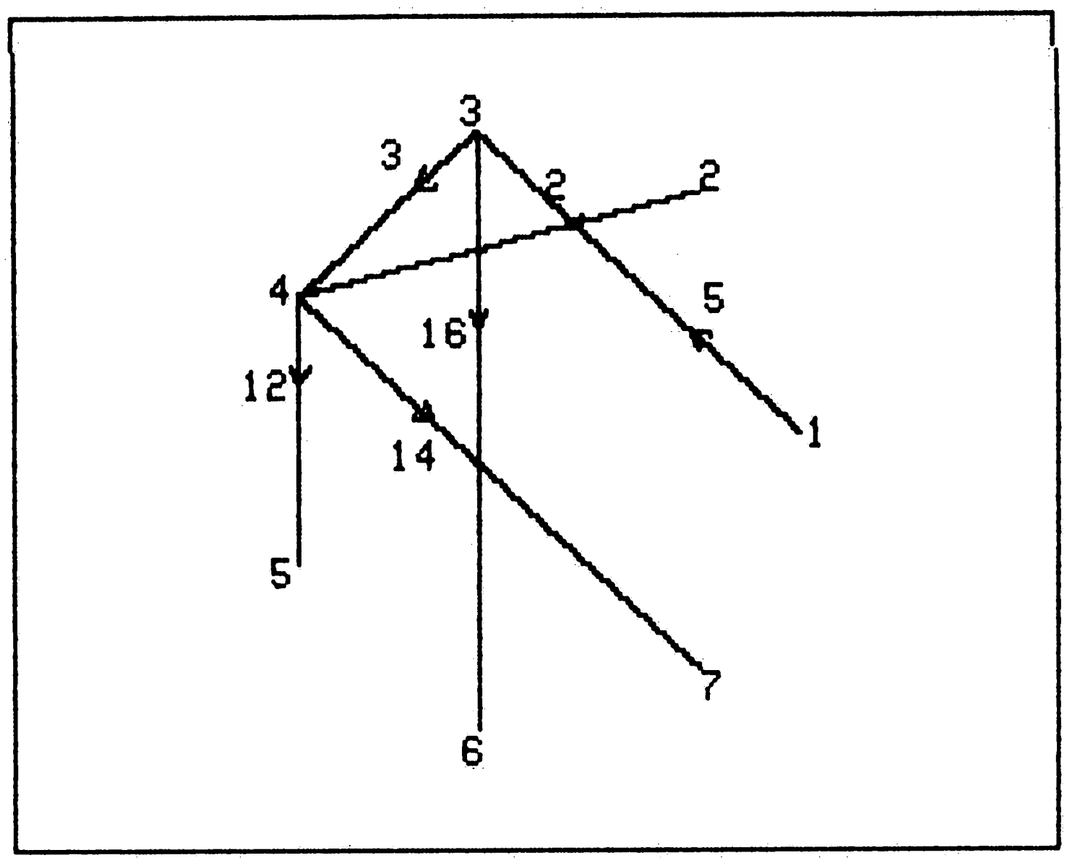
Figure 12: Sample Animation Frame 9

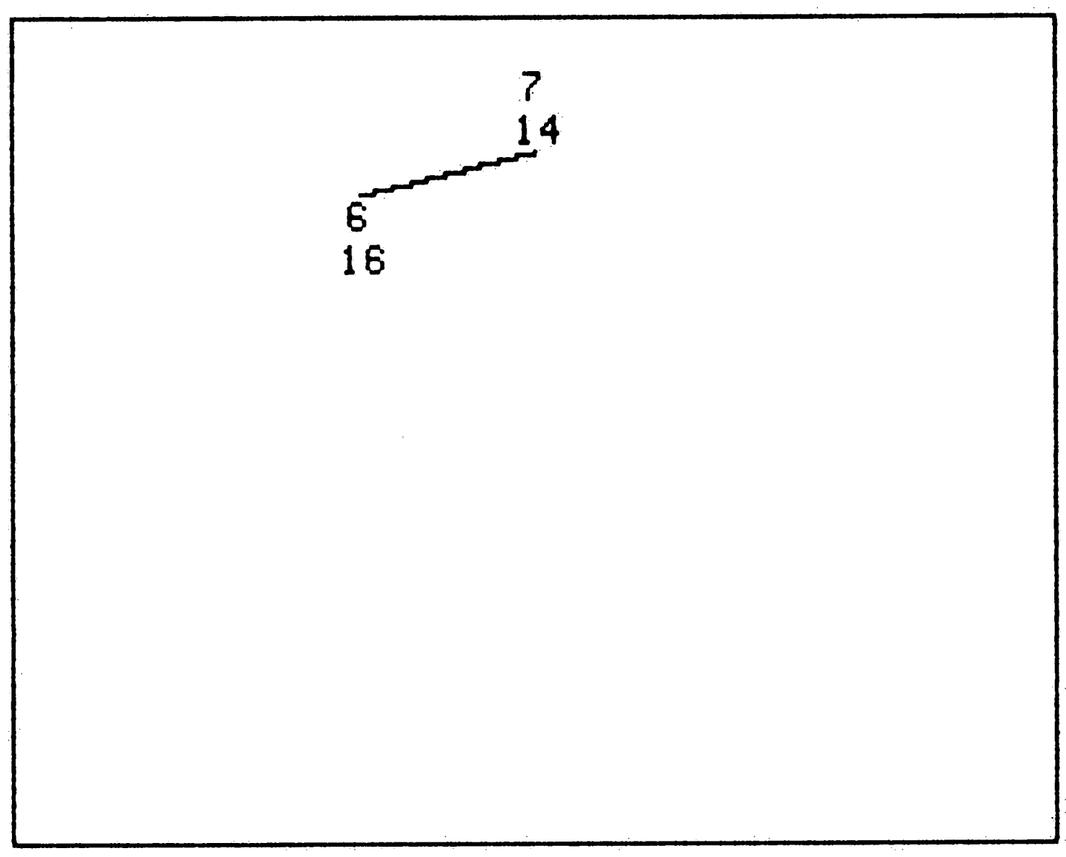





```
VERTEX 2
EDGE (2,5)
EDGE (2,4)
EDGE (2,3)
EDGE (1,2)
   enter DELETEMIN
   exit  DELETEMIN:=5
```

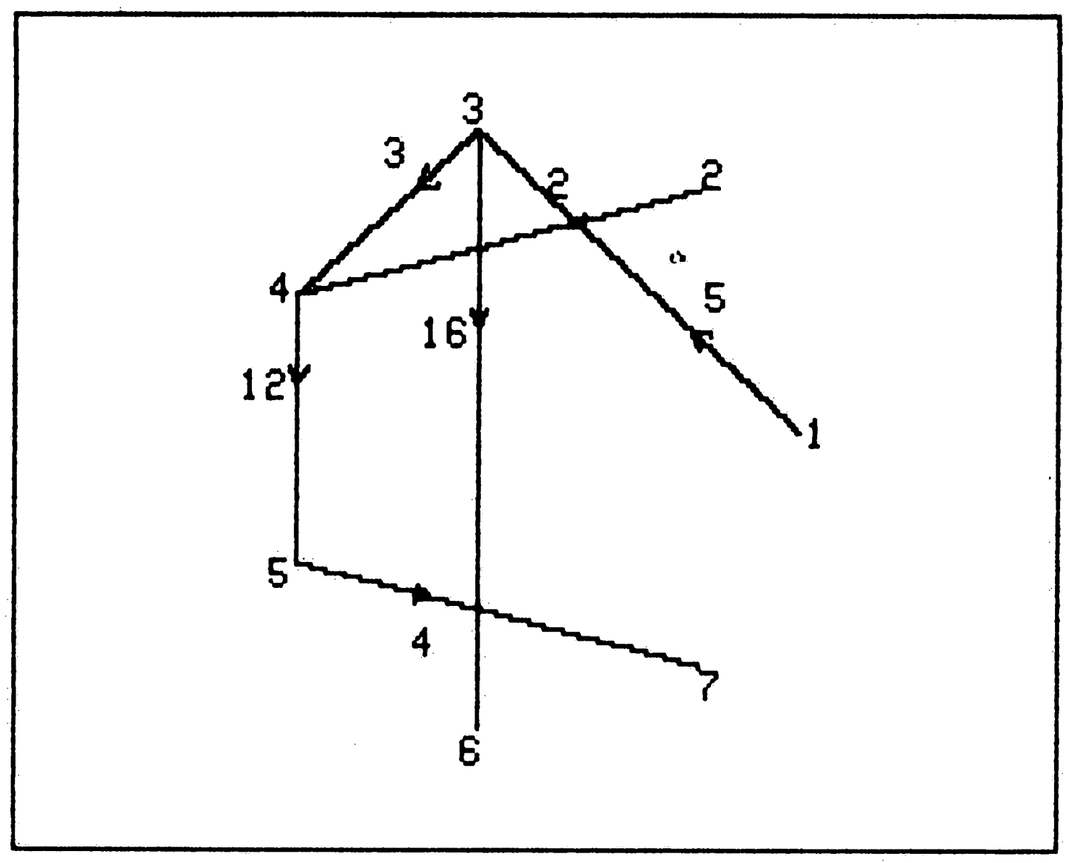
Figure 12: Sample Animation Frame 10

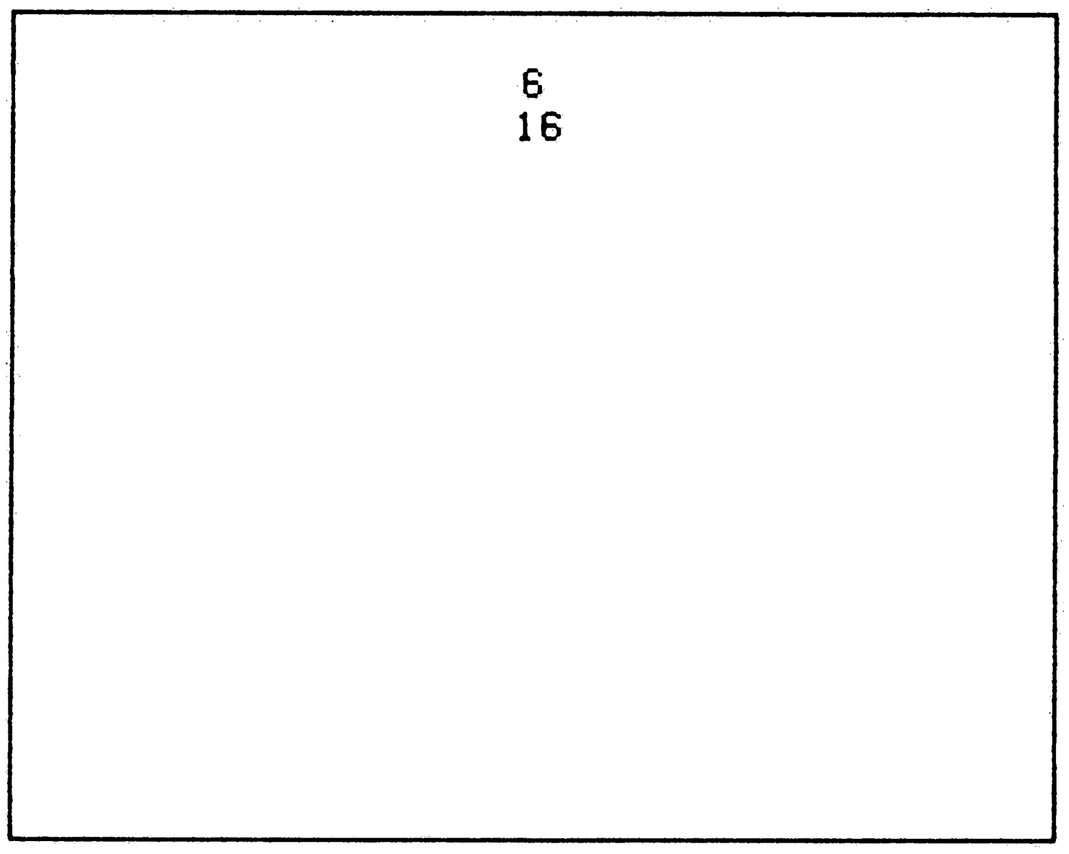





```
VERTEX 5
EDGE (5,7)
 CHANGING EDGE (4,7) TO RED
 CHANGING EDGE(5,7) TO BLUE
EDGE (4,5)
EDGE (2,5)
    enter DELETEMIN
    exit  DELETEMIN:=7
```

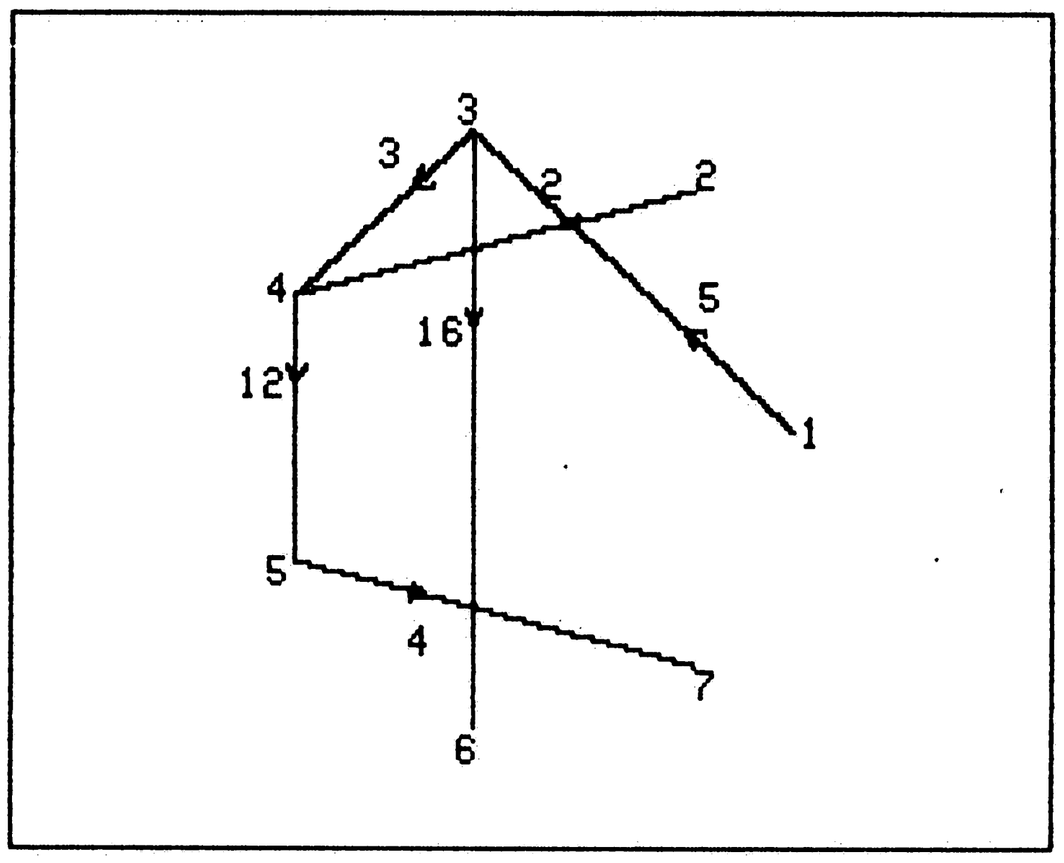
Figure 12: Sample Animation Frame 11





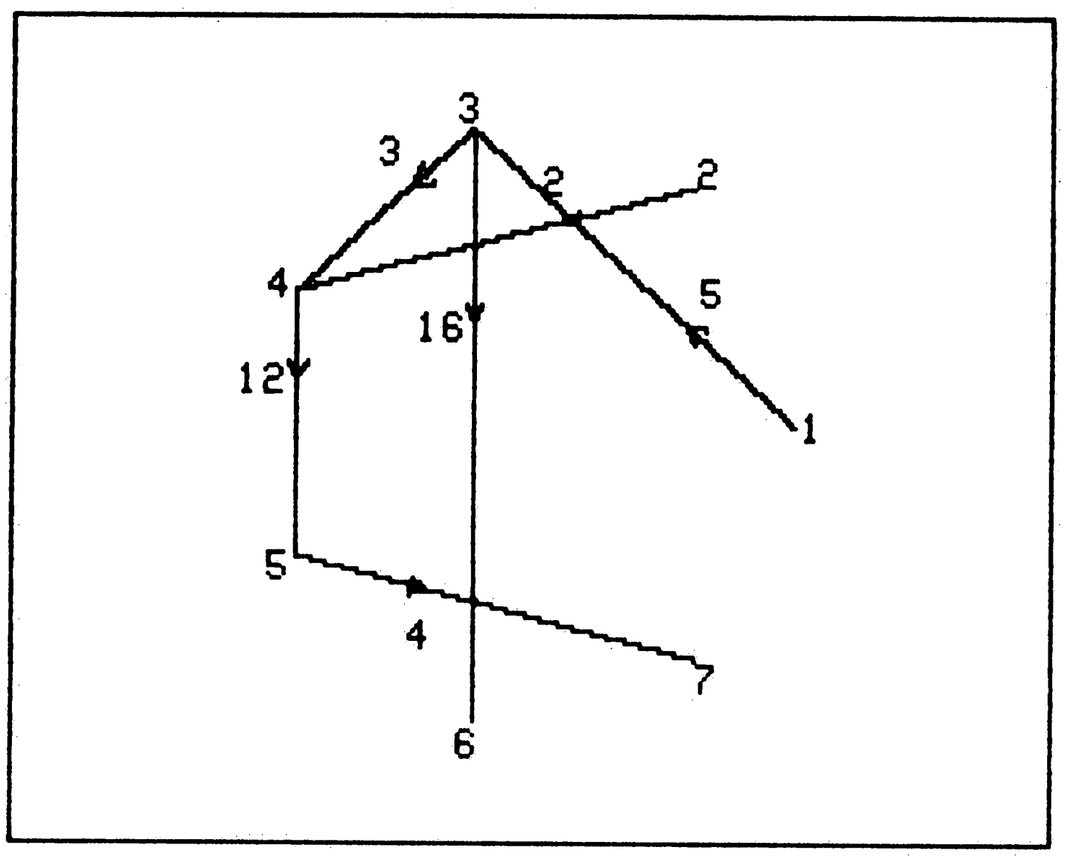

```
VERTEX 7
EDGE (6,7)
EDGE (5,7)
EDGE (4,7)
    enter DELETEMIN
    exit  DELETEMIN:=6
```

Figure 12: Sample Animation Frame 12




```
VERTEX 6
EDGE (6,7)
EDGE (4,6)
EDGE (3,6)
    enter DELETEMIN
    exit  DELETEMIN:=0
```

Figure 12: Sample Animation Frame 13



RESULTS OF PRIM'S MINIMUM SPANNING TREE ALGORITHM:

VERTEX	BLUE EDGE	COST	START	END
2	4	2	2	4
3	2	5	1	3
4	6	3	3	4
5	8	12	4	5
6	7	16	3	6
7	11	4	5	7

Figure 12: Sample Animation Frame 14

REFERENCES

1. Brown, M.H., and Sedgewick, R. A System for Algorithm Animation. Computer Graphics 18, 3 (July 1984), 177-186

2. Tarjan, R.E. Data Structures and Network Algorithms. Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania. 1983.

3. Hopkin, D. and Moss, B. Automata. North-Holland Publishing Company, Amsterdam, The Netherlands, 1976.

4. Culik II, K. A Model for the Formal Definition of Programming Languages. International Journal of Computer Mathematics Section A, Volume 3 (1973), 315-345.

5. Baase, S. Computer Algorithms. Addison-Wesley Publishing Company, Reading, Massachusetts, 1978

6. IMAGE Database Management System Reference Manual Part No. 32215-90003, Hewlett-Packard, Cupertino, California, 1983

7. Kamins, S. Applesoft BASIC Programmer's Reference Manual, Volumes 1 and 2.Apple Computer, Incorporated, Cupertino, California, 1982.

8. McCarthy, J., Abrahams, P.W., Edwards, D.J., Hart, T.P., and Levin, M.I., LISP 1.5 Programmer's Manual. M.I.T. Press, Cambridge, Massachusetts, 1976.

9. Myers, B, INCENSE: A System for Displaying Data Structures. Computer Graphics, 17, 3 (July, 1983), 115-125

10. Floriani, P.J. STREAMER - a program written for I&CP group CIM Systems of AMP Incorporated, Harrisburg, Pennsylvania. February-April 1984.

11. Control Data Cyber 70 Computer Systems models 72,73,74 6000 Computer Systems: SCOPE Reference Manual Models 72,73,74 Version 3.4 6000 Version 3.4, Control Data Corporation, Arden Hills, Minnesota, 1973

12. Dionne, M.S, and Mackworth, A.K., ANTICS: A System for Animating LISP Programs. Computer Graphics and Image Processing, 7 (1978), 105-119

13. Reiss, S.P., Graphical Program Development with PECAN Program Development Systems. Software Engineering Notes 9,3 (May 1984), 30-41.

14. Foley, J.D., and Van Dam, A., Fundamentals of Interactive Computer Graphics. Addison-Wesley Publishing Company, Reading, Massachusetts, 1982.

15. Rodgers, R. and Hammerstein II, O., Getting to Know You from: The King and I, Williamson Music Inc., New York, N.Y., 1951

Appendix A

Externals used for the stack animation of Figure 7

```
PROCEDURE newvariable(VAR vname,vtype:string);
{ creates an animation system based variable with
identifier 'vname' and of type 'vtype'}

PROCEDURE getvariable(VAR vname:string;VAR x:INTEGER);
{ returns the value of the animation system based variable
   'vname' in 'x'}

PROCEDURE setvariable(VAR vname:string;x:INTEGER);
{ changes the value of the animation system based variable
   'vname' to 'x'}

PROCEDURE newscreen(VAR id:INTEGER);
{ makes a window area available for display - it is
referenced by an identifier 'id' which is returned.  The
area is defined in the global variable 'screen'}

PROCEDURE standardheading(s:INTEGER);
{ inserts the standard heading information into the screen
id 's'.  See Figure 6 for details}
```

```
PROCEDURE boxstring(screen:INTEGER;VAR txt,str:string;
                    len:INTEGER;x,y:INTEGER);
{displays the value of the string 'str' of length 'len' in
a box on screen 'screen', with lower left corner (x,y).
a title 'txt' is displayed to the left of the box}

PROCEDURE boxinteger(screen:INTEGER;VAR txt:string;
                     int,len:INTEGER;x,y:INTEGER);
{displays the value of the integer 'int' (rightmost 'len'
digits) in a box on screen 'screen', with lower left corner
(x,y).  a title 'txt' is displayed to the left of the box}

PROCEDURE label(screen:INTEGER;VAR txt:string;x,y:INTEGER);
{displays the label 'txt' on screen 'screen' with lower
left corner (x,y)}

FUNCTION str(n:INTEGER):string;
{ converts 'n' to its string representation (in base 10,
with leading zeros removed) and returns it}

GLOBALS
stdchar.xsize     standard character x dimension in pixels
stdchar.ysize     standard character y dimension in pixels
screen[s].xsize   for screen number s: x length in pixels
screen[s].ysize   for screen number s: y length in pixels
```

Appendix B

The program "STREAMER"

This program is a user utility which performs time-delayed and scheduled submission of jobs. While it has access to "privileged" information, i.e. passwords, it uses a set of rules slightly more restrictive than the operating system to test for legal user access to the jobs. A unique feature is its ability to trace execution within the jobs it submits, by means of the command, "TELLSTREAMER".

When, in the course of execution of a job submitted by STREAMER, the operating system executes this statement, the text message to the right of the TELLSTREAMER command is sent to the main STREAMER program, which records it for access by the requesting user. This accomplishes a simple "animation" of the system domain, as a user may repeatedly request information on the last message sent from a particular job. Except for inter-process interrupts, no special or system-privileged functions were required to implement this feature.

The author wrote it while employed at AMP Incorporated, Harrisburg, PA, as System Software Specialist for I&CP group CIM Systems, during February and March of 1984.

Appendix C

Externals used by the procedures of Figure 11

```
PROCEDURE showcall(s:string);
{displays the line 'enter'+s at the bottom of the code
window, the top line is rolled off the top}

PROCEDURE showreturn(s:string);
{displays the line 'exit'+s at the bottom of the code
window, the top line is rolled off the top}

PROCEDURE showline(s:string);
{displays the line s at the bottom of the code window, the
top line is rolled off the top}

PROCEDURE drawedge(VAR g:graph;e,c,w:INTEGER);
{ draws edge 'e' of graph using color 'c' in window 'w'}

FUNCTION n2s(z:INTEGER):string;
{ returns the string form of 'z' }

FUNCTION parent(z:INTEGER;VAR h:dheap):INTEGER;
{ returns the parent node in dheap 'h' of node 'z'}
```

```
FUNCTION dminchild(z:integer;VAR h:dheap):integer;
{returns the child of node 'z' in dheap 'h' with the
minimum key value}

PROCEDURE linew(xf,yf,xt,yt,c,w:INTEGER);
{draws a line from (xf,yf) to (xt,yt) with color 'c' in
window 'w' }

PROCEDURE charw(x,y,K,c1,c2,w:INTEGER);
{draws the character with ASCII ordinal 'K' using color
'c1' on a background of color 'c2' in a 5 by 7 pixel cell
with its lower left at (x,y) in window 'w'}

PROCEDURE numw(x,y,n,l,c,w:INTEGER);
{converts the number 'n' to a string of length 'l' which is
drawn in color 'c' on BLACK with the lower left at (x,y) in
window 'w'}
```

BIOGRAPHY OF THE AUTHOR

Peter Joseph Floriani was born July 7, 1955 in Reading Pennsylvania to Basil and Elnor (Krebs) Floriani. He is the oldest of their eight children. He received his Bachelor of Science degree in Fundamental Science from Lehigh University in January, 1981. He was employed by Frankel Engineering Laboratories, Reading, Pennsylvania from September, 1977 to December 1983 as a systems programmer and analyst, concentrating on software design and implementation of both numerical control applications and system utilities for the Hewlett-Packard 3000 system. In January, 1984, he started working for AMP, Incorporated in Harrisburg, Pennsylvania, as system software specialist, handling system utilities and internals for I&CP group CIM Systems. After a corporate reorganization in September, 1984 created an unfavorable situation, he left AMP, and was admitted to graduate study at Lehigh in January, 1985. Having passed his first 12 hours with a straight 'A' average, he expects to complete the remaining coursework by December, 1985, at which point he will evaluate continuing for the PhD. He has been a member of the Association for Computing Machinery (the ACM) since 1982, as well as several Special Interest Groups. He is an active member of the Beta Theta Pi Fraternity, which he joined at Lehigh.